Economic Allocation of Computation Time with Computation Markets

by

Nathaniel Rockwood Bogan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

© Nathaniel R. Bogan, 1994

The author hereby grants to M.I.T. permission to reproduce and to distribute copies of this thesis document in whole or in part, and to grant others the right to do so.

Signature of Author ..... Nathaniel R. Bogan .....
Department of Electrical Engineering and Computer Science
May 15, 1994

Certified by .....
Jon Doyle
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Department Committee on Undergraduate Theses

WITHDRAWN
Eng.
FROM
MASS. INST. TECH.
MIT LIBRARIES
JUN 15
LIBRARIES

Economic Allocation of Computation Time with Computation Markets

by

Nathaniel Rockwood Bogan

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 1994, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The use of economic principles in computer science is a comparatively recent development. The application of economic concepts such as marginal utility, preferences, efficiency, and resource allocation have begun to be discussed at a theoretical level. Several systems have been developed that attack problems of computer science using these principles. The applications include use of idle resources in a network and approximating solutions of complex problems by transforming the problem into a general equilibrium framework. This thesis uses market mechanisms more directly and addresses the problem of allocating processor time using a market in computation. This allows for efficient use of processor time, and is especially useful in the case where many more useful tasks could be performed than time constraints may allow. The unique problems associated with a market for a processor are discussed. A protocol is proposed to rent the processor at a rate per-time-unit, determined by market forces.

Thesis Supervisor: Jon Doyle
Title: Principal Research Scientist

Acknowledgments

I would like to thank:

- Jon Doyle, my thesis advisor, for introducing me to the field and to graduate research. I am also thankful for his flexible nature, his strong support, and his careful critiques of numerous drafts of the thesis.

- Michael Wellman, who perhaps planted the seed of economics into the Medical Group at MIT, and whose WALRAS system serves as a platform for all of my coding. I am especially grateful for his helpful advice and willingness to answer questions during the technical development stages.

- Eileen Brooks for serving as a sort of economics advisor and a sounding board for ideas in the application of economics to computer science. This work might never have gotten off the ground without her help.

- My mother Elizabeth for further advice concerning economics but most of all for her undying belief that I would succeed at anything I laid my hand to.

- My father Thomas for contributions concerning risk and futures markets, and for his steady support throughout my life.

- My brother Andrew for invaluable assistance in clarifying my explanations and helping me with accessible examples.

- Bobbie-Jo Hutchinson, who by the time of publication will be my wife, for continual emotional support and for making my years at MIT so much more enjoyable.

- All of the brothers of Theta Chi fraternity, Beta chapter, who have helped to shape me into the man I am today.

Contents

Chapter 1

Introduction

For many applications in computer science, the availability of processor time is a significant limiting factor. It is normally desirable to perform tasks as quickly as possible and to utilize approaches or algorithms that are most applicable and efficient. Trade-offs among different methods and resources must also be considered. The final goal is a rational allocation of computation time and other computational resources.

In this thesis, I explore the issue of allocating processor time by using a market-based approach. Markets and the science of economics offer many possible advantages, which are discussed in chapter 2.

The task of guiding automated reasoning presents an especially appropriate domain for the market-based approach, and my trial implementations have centered on this problem. In the following section I briefly introduce control of reasoning as a natural case for an economic approach to allocation of computation time.

1.1 Control of Reasoning

It is a principal goal of artificial intelligence to mechanize logical reasoning and rational decision-making. For many approaches to this problem, it is useful to represent what the agent believes about the world in which decisions are being made. The reasoner

maintains a set of beliefs from which to make inferences, draw conclusions, and decide actions. All of these activities may result in new beliefs, and provide new information, possibly contradictory, regarding old beliefs.

The task of guiding revision of the reasoner's beliefs is thus, in itself, an important part of controlling reasoning based on beliefs. At a high level of abstraction, there are two approaches to maintenance of the reasoner's beliefs. The first is to insist that all beliefs are consistent and all inferences sound. The principal advantage to this approach is that there is never any doubt about justifications. All inferences, conclusions, and decisions will be provably valid, in the sense of mathematical logic. However, requiring total coherence may prove intractable with large numbers of beliefs and frequent changes to these beliefs.

Consider a person watching a football game. At one point she notices that Ronnie Lott is wearing a Jet uniform, contradicting her belief that Lott plays for the Raiders. To be certain that her beliefs are still sound and consistent after the introduction of this new belief, an extremely complicated search of a massive database (her brain) must take place, propagating this update. Intuitively, this seems to be a large waste of time, especially considering what a small portion of the brain is devoted to beliefs concerning football. This intuition motivates a second approach to maintaining beliefs.

The guiding principle of this second method is the rational use of time and other resources in maintaining beliefs. Rationality in this context refers to the economic definition, namely the optimum use of scarce resources under (possibly) incomplete information. Many models in artificial intelligence use this principle implicitly by using algorithms designed to avoid unnecessary calculation. This is often called "efficiency" by computer scientists, where efficient in this case is usually translated to fast (relative to the next fastest method). Economists have established a wealth of research concerning a more general notion of efficiency resulting from equilibria.

In this research, the principles of economic rationality and efficiency are employed more directly. An efficient allocation of resources (in particular computation time) is

achieved by using a market economy as the underlying computational model. Processes act as agents in a free market, competing for the use of limited resources. In light of the recent downfall of totalitarian communism, it seems ironic that the computation model in which each process is explicitly instructed what to do when has received virtually all of the attention of computer scientists.

In fact, free markets and decentralized decision-making offer most of the same advantages in computation that they do in the business economy. Of great importance are tremendously increased sensitivity to world dynamics, especially locally, and the ability to control processes much more efficiently through specialization of methods and information. (For a detailed discussion of the advantages of decentralized decision-making, see [2].) Perhaps most importantly, markets can be designed to provide maximally efficient allocation of resources, of critical significance for automated reasoning and many other applications in computer science.

1.2 Guide to the Thesis

The principal contributions of this thesis are a definition of economically efficient use of time, and description of a market mechanism that ensures such an allocation. To glean the most significant discoveries, the reader need only read chapter 4 and section 5.5.2 However, I have also provided motivation for my approach, and discussed many alternatives, issues, and open problems. The thesis is organized as follows:

Chapter 2 provides an abstract justification for the use of economic principles in computer science. Readers not familiar with the merits of this approach may find the rest of the thesis confusing if they have not read this chapter.

Chapter 3 outlines important contributions of previous research that relates economic principles to computer science. No material in this chapter is a prerequisite for other chapters.

Chapter 4 describes the goals of processor allocation and defines much of the termi-

nology used. It outlines why a processor market is non-trivial, and provides a few general classes of applications. Chapter 4 provides much of the groundwork for the remainder of the thesis.

Chapter 5 critiques a variety of possible models for a market in processor time, including the successful processor rental model discussed in section 5.5.2.

Chapter 6 describes the computational market that I have implemented as part of the RECON reasoning economy. These details are likely only significant to the reader who intends to build a market-based computation system.

Chapter 7 looks at many of the problems to which my research has opened a door. It discusses the limitations of computational markets, and discusses many issues that are not completely solved by my research. It further addresses potential difficulties in practical implementations and scaling of computational economies to larger systems. It also includes two possible extensions of the processor market mechanism to a memory market and to a multi-processor market. Readers interested in opportunities for further research in related areas will certainly want to read this chapter.

Chapter 2

The Role of Economics in Computer Science

The use of economics in computer science is not a completely new idea. Economic principles such as rationality and efficiency have been used implicitly in artificial intelligence for many years. Only recently, however, has the science of economics been used directly as a guide for research in computer science. The systems and research described in chapter 3 serve as an empirical justification for the use of economics in computer science. In this chapter, I provide a more theoretical justification, and describe some of the potential benefits offered to computer science by economics.

Because a large body of work already exists in economic theory and analysis of real-world economies, the possibility exists that a great deal can be learned from this science if it proves applicable. In addition, economics provides well-established and well-defined terminology which may help to provide standardization for similar notions in computer science.

This chapter provides only the highlights to serve as goals and general justification for applying the economic methodology. For additional details of the possible advantages of economics see [22, 21, 10, 30, 2].

2.1 Resource Allocation

One of the principal topics in economics is the allocation of resources. Substantial research has been conducted both in how resources are allocated in human society, and how they should be allocated. Specifically, economists are interested in allocating *scarce* resources. A resource is scarce if the demand for the resource exceeds the supply when the cost is zero. This will be the case for most resources that are considered "desirable." Examples of scarce resources include virtually all of the products that we buy from day to day. One example of a resource that is usually not scarce is air, since, under normal circumstances, air is free and the supply exceeds the demand. In a computer, most resources will be scarce, since if there were no cost whatsoever (including opportunity costs from *not* doing something else), processor time, memory space, and disk space will be in greater demand than their supply. In general, this cannot be corrected by simply increasing the supply to a new fixed value. Countless computer programmers who once claimed they would never need more than 256k are now pushing the limits of their 16 Megabyte machines.

Economists have established a well-tried solution to the scarce resource allocation problem, namely the use of **markets**. A great volume of research shows that markets can achieve optimal allocations under many circumstances with little or no central guidance. Each market participant need only think of itself[1], and the equilibrium allocation will very often be optimal (in the sense that it could not be improved upon by an omniscient resource allocator), or very close to optimal. Markets also have significant advantages over central allocation schemes, some of which are mentioned in the following sections.

In light of this research, it seems only natural to use market mechanisms to allocate resources in a computer.

[1] I use the neuter form for participants so as include organizations, individuals, and computational entities.

2.2 Efficiency

Economic theory also provides a definition of efficiency that is in some cases more precise or more useful than its common meaning in computer science. Computer scientists usually use the term "efficient" to mean fast compared to other possible methods. This is the standard meaning of an efficient algorithm in computer science. Because efficient is defined relative to other options, it often has an imprecise meaning since there may exist faster options that are unknown.

Economists generalize the meaning of efficiency to include not only speed (the "efficient" use of time), but efficient use of all other resources as well. Economic efficiency is concerned with the tradeoffs that exist between resources, the competition for their use, and conditions which imply that all resources are simultaneously being used for maximal benefit. One important criterion for efficiency is the Pareto optimality condition. An allocation is Pareto optimal if any possible trade would make at least one agent worse off. Intuitively, this condition means that all agents involved are maximizing their benefit.

This generalized notion of efficiency can be extended to discuss "rational" allocations and "rational" decision making. The Pareto optimality condition specifies that given the preferences of the agents involved, the allocation cannot be improved upon in every aspect. Thus all Pareto optimal points can be considered "rational" since deviations cannot result in strictly superior "solutions."

2.3 Decentralized Decision Making

Market systems are based on decentralized decision making. Each participant needs only local information with which to make decisions concerning preferences and eventually valuations of goods and services. It many cases it may be much easier to formulate solutions to local decisions than to global ones. It is also very likely that there may be too much information for a central decision maker to process in any reasonable amount of time.

In any situation in which there is a tremendous amount of total information and/or a very high rate of incoming information, it may be impractical or impossible to make sensible decisions centrally. In the case of uncertain or conflicting information, decentralized participants may have a much better chance to make a reasonable choice, since they will presumably have less uncertainty and fewer contradictions to deal with. Since many applications of artificial intelligence deal with large amounts of information, which may be uncertain or conflicting, this advantage may carry over into computer science.

Decentralization may also allow for specialization to local problems. Individuals in the same "business" may develop more efficient means of communicating and exchanging knowledge that are specific to their business. A central decision maker would have to know all of these "languages" in order to process the available information.

Of course, this should not be viewed as a guaranteed route to success. In many cases, central planners *will* be able to operate more quickly than decentralized planners and still make good decisions. This will especially be the case when the solutions to the problems at hand are well known and the system is serving more as a command hierarchy than a decision maker. For example a computer that is asked to evaluate $2 * (4 + 7 - \sqrt{8 * 9})$, probably doesn't want to spend any effort determining what to do first. A fast method for doing this evaluation is already known.

For a more thorough discussion of the possible advantages of decentralized decision making, see [2].

2.4 Dynamic Adjustment

One of the greatest strengths of well designed markets is their ability to adjust very rapidly to sudden, unforeseeable changes. Each participant need only adjust its own valuations based on the changes, and the equilibrium will reflect the new optimal allocation of resources. No authority is needed to determine a new solution - the market provides that automatically.

This is a significant advantage in a computer system, especially in artificial intelligence or in control systems. The environment is constantly changing and in general it will be impossible to predict what will happen in the future. Rapid adjustment to these changes is highly desirable.

One unfortunate side effect is that markets are very often chaotic systems in the sense that very slight variations in parameters may result in very large changes in the future course of the market. Dynamic adjustment often demands this property, but it can make it very difficult to analyze the behavior of a system or predict how it will act in the future.

2.5 Sensitivity to Demand

Markets have a remarkable ability to produce those things that are demanded, even if they are not what a central analyst would have guessed. This may allow a computer system based on markets to make surprising decisions and appear to "invent" solutions simply because the demands of the participants have evolved over time.

An interesting real-world example of this behavior, which unfortunately will date this thesis, is the production of Carolina Panther hats in 1993, even though the Carolina Panthers will play their first National Football League game in 1995. It seems unlikely that a central planner could have anticipated such a strong demand for these hats at this time. But the market mechanism met the demand almost immediately.

On the other side, an often cited problem with the former Soviet command economy was that production was based on predetermined goals, and did not necessarily meet any demand. In one case, this meant that a factory produced literally tons of nails that were all exactly the same, even though users of nails really needed much more specialization.

2.6 The Success of Real-World Markets

Finally, it makes some sense to look at market decision-making systems and central command systems in the real world. Although it is impossible to separate other influencing factors with certainty, it seems that relatively unregulated market systems like those found in the United States and South Korea have drastically outperformed command economies such as the former Soviet Union and North Korea.

Optimistically, it may be possible to achieve similar outperformance of central command systems in a computer.

2.7 Tradeoffs

Of course, using economics in computer science is not without its tradeoffs. Several possible problems have been mentioned earlier, including chaos (mathematically speaking) and overthinking of simple problems. In a computer system, markets may also be difficult to structure, and the constraints on programming methods may be significant. It may also be very difficult to determine the preferences (and thus utility functions) of participants, and it is very unlikely that any particular set of preferences can be justified on a theoretical basis. These issues, as they relate to processor allocation, are discussed in more detail in chapter 7.

Perhaps most important is the tradeoff of time and resources spent *operating the market* versus the improvements in performance (if any) gained from its use. In particular, in the WALRAS-based processor market that I have implemented, the system spends on average over 10 times as long computing market equilibria as it does executing processes. Of course, this is something of a "toy" example in which the processes are very simple. In addition, no part of this system is designed for speed, and presumably market mechanisms can be coded into hardware (possibly with a dedicated processor), resulting in drastic speed improvements. Finally, real world markets do not "compute" equilibria but evolve to them over time. If time spent computing equilibria is too great, this solution may

be used in a computer system as well. Even without those improvements, the benefits of the market should outweigh the time "wasted" in operating it when the tasks to be performed are large relative to the market operating time or the benefits outlined in this chapter are highly desirable.

Chapter 3

History of Related Work

Several substantial works have utilized economics for solving problems of computer science or have paved the way for its use in the future. In this chapter, I briefly describe some of the relevant contributions of five significant works.

3.1 The Contract Net

Randall Davis and Reid Smith developed an extensive model of communication between decentralized agents called the Contract Net[2]. Although the contract net is not a market, per se, the work provides significant grounding for decentralized agents to participate in a coordinated way. The distribution of tasks is viewed as a form of contract negotiation. Because negotiation and exchange of information are often critical to the success of a market, their work is very relevant to the development of complex market mechanisms in which communication between participants is required.

In a canonical case, the contract net protocol proceeds in the following steps:[1]

- *Task announcement.* When a particular agent has decomposed a task into subtasks or has realized that its resources may not be ideal for completion of the task, it issues a *task announcement* and serves as the *manager* of that task.

[1]See section 7.3 of [2] for a complete description of the contract net protocol.

- *Task evaluation.* Other agents in the system may examine the task announcement and determine if they have interest in the task and access to some or all of the resources necessary to complete it.

- *Bidding.* Agents that are interested in an announced task submit "bids." These are not bids as in an auction, but refer to a set of capabilities and qualifications relevant to the task.

- *Bid evaluation.* The manager of a task evaluates the bids and determines to which agent(s) to award the contract.

- *Awarding.* The manager communicates an award message to the selected agents. These nodes are termed *contractors* for the task.

Contractors may then become managers of more finely divided subtasks, forming an arbitrarily deep hierarchy as needed.[2] Contractors may also communicate various progress reports to the task manager as well as a description of results and a notice of completion of the task.

By the mechanism of contract negotiation, the contract net allows for sophisticated communication and coordination among processes which are under no central control.

3.2 Enterprise

Thomas Malone's Enterprise task scheduler [19] employs many of the concepts of the contract net. The system is designed to schedule tasks in a network of personal workstations. Rather than using the traditional view that each station be used only by its "owner," Enterprise attempts to match pending tasks with the most compatible resources available at run-time.

[2]There is also nothing preventing the manager of a task from later becoming a contractor for some subtask.

This methodology increases the inherent power of a distributed network because the machines are allowed to work together to a much greater extent. Malone found that significant performance improvements were realized in systems with about 5 to 10 workstations, though the benefits leveled off for larger networks.

One of several reasons for the observed improvements is that the system is able to take advantage of unused processing power. When a station's "owner" does not have any pending tasks, the station will remain idle in a traditional network system. Enterprise provides a way to take advantage of this idle time by scheduling pending tasks from other machines or executing lower priority tasks on the otherwise idle processor.

Malone further finds even under very unreliable information concerning estimates of processing time and resource demands, as well as task priorities, that the Enterprise system usually allows for noticeable improvements over standard network management techniques.

However, the Enterprise system does not use a market to determine the relative values of different tasks. In fact, these values, which Malone calls "priorities," are determined by a variety of heuristics (including length of execution time), that to a certain extent are arbitrary. See section 5.1 for more discussion of priorities based on length of execution time alone.

The Enterprise distributed scheduling protocol proceeds with the following essential steps:[3]

- *Request.* A client (similar to a manager in the contract net protocol) announces a task with the level of priority, description of required resources, and enough information for machines to estimate processing time.

- *Response.* Unused machines "bid" by giving their estimated processing time for the task. Busy machines form a queue of processes on which to bid when they complete their current task.

[3]See [19], figure 2 and surrounding discussion, for a complete description of this protocol.

- *Task assignment.* The client evaluates bids and awards the task to the most appropriate machine. If no bids are received then the task will be awarded to the first bidding machine.

In addition, a client may cancel a task request at any time, and is required to do so when a contractor delivers a task completion message.

In summary, the Enterprise system provides a greatly improved method of utilizing resources and responding to the changing needs of a network's users as a whole.

3.3 SPAWN

Carl Waldspurger et al. have implemented a system called SPAWN [29] that is more directly based on market forces for allocating resources in a distributed network. The system allows much greater capacity for tradeoffs between resources of different quality than does Enterprise. This allows more efficient resource allocation (in an economic sense). For example, a fast processor will normally have a higher price than a slower one. Tasks that do not need much processing speed can pay less for their computation than those requiring more processing power.

SPAWN provides the first framework for an actual market in computational resources. The relative values of different resources are determined in a market, and relative importance of tasks is determined primarily by funding: tasks with more funding will be able to outbid those with less and thus buy the most desirable resources.

SPAWN provides an elegant solution to the problem of scheduling posted tasks on a distributed, homogeneous network. It builds on Enterprise and the Contract Net, and allows market forces to determine relative prices (and thus implicitly relative value or importance) of resources. SPAWN also provides significant practical evidence that a computational economy can in fact offer many of the benefits of economics that were espoused in chapter 2.

However, because SPAWN is designed as a task scheduler, it does not use markets to

determine the relative merit of different tasks. Instead, task have sponsors, and, much as in the escalator algorithm ([21]; also discussed here in section 5.7), these sponsors can provide more and more funding as time passes and the task has not yet executed.[4] The use of pure markets to determine relative valuations of tasks might result in some tasks never executing because they are always outbid. In a standard task scheduler, this is very undesirable. See section 4.3.2 for a complete analysis of the difference between task scheduling and efficient processor allocation.

Furthermore, Waldspurger's paper leaves the decision concerning how to evaluate processes' bids entirely up to each machine. Although the distributed network has several additional complications, such as the need to promptly return control of a workstation to its human owner, some of my research may apply to the problem of entering and choosing among bids for processing resources, even in this rather different application.

3.4 WALRAS

Michael Wellman's WALRAS architecture [30] provides a means for creating general agents in a computational economy using the Common LISP Object System. My implementations are all built on the WALRAS mechanisms, and are discussed in chapter 6. In this section, I describe in more generality some of Wellman's accomplishments.

"WALRAS is a general 'market-oriented programming' environment for the construction and analysis of distributed resource allocation systems, based on general equilibrium [30, abstract]." Its primary intended purpose is to allow complex distributed problems to be transformed into a general equilibrium problem in economics. WALRAS can then be used to compute the solution to the original problem by computing the general equilibrium.[5]

WALRAS provides a means of defining various parts of an economy. A *good* represents

[4]This represents a gross over-simplification, but is sufficient to understand the basic mechanisms.

[5]The method used to solve for the equilibrium is a variation on a method suggested by Léon Walras, a French economist after whom Wellman's system is named.

anything that can be traded, including services performed by processes. An *auction* represents a mechanism for accepting bids and determining prices for a specific good. *Agents* can be any sort of entity which will participate in an auction. *Consumers* are a broad class of agents who participate by maximizing certain supplied utility functions. Consumers are usually endowed with some amount of various goods, which they attempt to trade so as to maximize utility. *Producers* are a broad class of agents which participate by maximizing profit from transforming certain inputs into certain outputs (which is meant to include services). *Bids* are entered by any agent participating in an auction, and are generally divided into *supply bids* and *demand bids*. These bids are given as a partial function of quantity demanded (or supplied) versus price.

WALRAS computes the market equilibrium for all auctions taken together in an atemporal fashion. Specifically, it computes only the equilibrium prices from the current bids. The system does not directly account for any sort of ongoing market activity. However, it can be applied repeatedly to achieve a discrete approximation to a continuous market system over time. The current implementation of the RECON system [12] employs this methodology.

WALRAS also makes an assumption often called "perfect competition" in economics. Specifically, each agent considers prices as "given," neglecting the effect of its own actions on the equilibrium price. This assumption results in increasingly optimal individual behavior as the number of participants in each auction is increased. However, the assumption does not allow for monopolies or small groups of suppliers to exploit "market power," which often would mean the ability to increase profits by increasing price. Though agents in this situation will not be acting in a completely rational way (i.e. acting to maximize individual benefit), there may actually be some advantages to eliminating market power. In particular, no anti-trust regulations like those used in real-world economies should be required.

Wellman has successfully used the WALRAS system to solve distributed transportation resource allocation problems, but I will not provide those details here.

3.5 The Work of Drexler and Miller

Mark Miller and Eric Drexler have written several papers [20, 22, 21] that justify the use of economic methods on theoretical grounds and provide a variety of specific market-like mechanisms. In particular, [22] provides a careful analysis of the use of economics in computer systems at a general level. Miller and Drexler identify many of the similarities and important differences between markets composed of human agents and those to be used in a computer. They discuss fundamental issues such as programmability of market objects, availability of information, ownership, security, and currency. The paper also discusses possible methods for and benefits of intertwining computational markets with human markets.

In [21], Miller and Drexler provide a detailed description of a market mechanism to determine processor use in scheduling on a single-processor machine. They identify constraints on market mechanisms, and suggest a specific means of auctioning the processor. The so-called escalator algorithm in discussed here in section 5.7, and the fundamental difference between the the problems attacked by Miller/Drexler and those of this thesis are described in section 4.3.2.

The paper also provides an elegant market-based storage management system, including garbage-collection guided by market forces. The analysis of this allocation system is very thorough and is beyond the scope of this thesis. However, a highly simplified discussion of storage rental is provided in section 7.2.

Since many of their observations, discoveries, and suggestions are directly relevant to this thesis, I have incorporated them directly into the text with appropriate references, and will not discuss them at length here.

Chapter 4

Goals of Processor Allocation

Despite continuing advances in hardware speed and power, it is very often an important goal in computer science to make the most of limited time. In this chapter I present a formal definition of the optimal use of processor time, based on the economic concept of efficient allocation as well as principles of rational decision-making. I also detail the goals of a computation market which is used to allocate computation time, discuss what makes the processor time market a non-trivial problem, and highlight a few general classes of applications.

4.1 The Value of Goods and Services

In order to use market forces in allocating processor time, it is necessary to determine the relative values of goods present in the computational economy. (As used herein, a *computational economy* means an economy in which computational resources and goods are traded.) These goods may refer to specific values that can be computed or to any other service that can be performed. Resources such as memory and disk space may also be treated as goods, but are generally more complicated because they tend to be used over some period of *time*. (See [21] and section 7.2 for more discussion of possible market mechanisms for these goods.) The processor itself is also a good, but has many

properties that make it rather different from the services performed by processes. These differences are discussed in the next section.

The computational economy must include buyers and sellers of goods with specific preferences and production abilities so that relative values of goods may be determined. There is no need to centrally assign values to any goods – the market equilibrium will determine all of their relative values from their availability and demand. Throughout this thesis, I will use the terminology used in Wellman's WALRAS computational economy (see section 3.4) to refer to the various agents and mechanisms of the economy.

The market used to determine processor prices and allocations is directly a part of this computational economy. The demand for services that can be performed (requiring processor time) determines the market value of processing time. Symmetrically, the cost of computing will affect the cost of performing these services, and thus their market price. The economic general equilibrium determines all of these heavily interdependent trading ratios (prices). Therefore, in the absence of an economy for other goods and services, a market for processor time would not be meaningful.

In addition, it is convenient notationally to introduce a numeraire good whose price is always equal to 1. This good serves as a sort of monetary unit to express relative prices in terms of a common unit. To preserve the parallel between the numeraire and real-world currencies, it is required that no consumer explicitly value the numeraire except as a means of acquiring desired goods. The actual value of the monetary unit will be determined implicitly by the initial endowments given consumers. If every consumer's endowment of the numeraire good is doubled, for example, the prices of all goods (when expressed in terms of the monetary unit), will also be doubled. There is no need for the monetary unit to represent any specific buying power; its relative value will be determined in the market just like the relative values of other goods. The purpose of the monetary unit is to facilitate description of prices and to allow all transactions to take place in the same denomination. The specific name of the monetary unit is, of course, of no significance. When constructing examples involving specific values I will use the standard

US currency $ notation to refer to the value of that good relative to the monetary unit. For example, if the service of calculating the square root of some variable x has a relative trading value equal to two units of the numeraire good, I will denote the price of calculating $\sqrt{x}$ as $2. This is purely for notational convenience; these price have no connection whatsoever to actual U.S. currency.

An important and very difficult question is how the various agents in the economy (especially the consumers) will determine their preferences. The consumers in the economy, taken as a whole, will be the directors of the systems. Their demand for goods and services will determine what has market value and thus which goods are produced (usually meaning computed) and which services are performed. Determining consumer preferences and initial endowments of goods (including the monetary unit) is an unsolved problem, but is not the subject of this thesis. Throughout the remaining discussion of the goals and problems of markets in processor time, I will assume that consumers and producers exist and can be used to determine the value of goods and services in the economy. I use the terms *processor market* or *computation market* to refer to the bidding, auctioning, and awarding mechanisms associated with the processor itself, but assumed to be part of a larger economy. I briefly return to the problems associated with designing market participants in chapter 7.

4.2 Special Considerations for Processor Markets

Although a processor market is similar in many ways to existing real-world markets, it has several critical properties that taken together make it rather different from conventional markets.

- The processor is unique. Only one process may compute at any given moment. (This constraint is relaxed in a parallel-processing machine, but there is still a fixed finite supply of computation.)

- The use of the processor is extremely time dependent. The past cannot be altered and only the future use of the resource can be determined. The resource is also quantified in terms of time. The consumer of computation uses a certain amount of *time*, which seems to indicate that prices and bids should be per unit time.

- There may be a minimum amount of computation time which has utility to a process. This is the case when a computation does not produce valuable intermediate results. It may be possible to partially sidestep this problem using some variation of *anytime* algorithms[4, 16, 32], but it also may be desirable to *require* that there be no intermediate results, for reasons discussed in section 7.4.3.

- The actions of a particular user of computation may drastically alter the needs of other market participants. The result of one computation may imply that a different computation is now considerably more or less useful than it was before. As a simple example, there may be several processes with a goal to calculate a particular value. Once that value has been calculated, their bids to use the computer for that purpose will be withdrawn. There is also the general possibility that a particular use of computation has positive externalities (see section 7.6.3) that may not be accounted for automatically by the market.

- Computing *now* is usually better than computing *later*. This is a very strange form of future devaluation. Unlike many real-word goods, which physically deteriorate or become technologically inferior with the passage of time, the processor will seem to be just as good when the future actually arrives as it is now. Still, it will often be more advantageous to use the processor now than to wait and use it in the future.[1]

- Processes may be either suspended or terminated if they are interrupted (presumably by a very high bidder). Determining which of these to do may be very difficult and may even place constraints on the types of computations which are allowed.

[1] It is also conceivable that the processor may actually be *more* valuable in the future, especially in the case where new technology is anticipated.

- There is not an a priori reason to assume that the market must be "fair" in any way. As an example, consumers may arbitrarily create bidding power (money) at any time, if it is allowed in the model. The computational agents are very much unlike human agents in this regard.

- Similarly, computational markets do not require many of the regulations of real-world markets. There is normally no concern of overworking, oppressing, or bankrupting agents in a computational economy. Regulations that *are* imposed may be considered more like physical laws than legislation, since the system can be designed so that agents simply do not violate certain rules.

The first two differences, uniqueness and time-dependence, cause the most separation from normal markets. In most markets, the good is relatively divisible (into individual oranges or gallons of fuel oil for example), and the buyer will simply take home however much (many) he or she is willing to pay for at the current price. A processor market does not work this way. In terms of real-world analogues, it is perhaps more like having control over time itself: A process can do nothing when it does not have control of the computer. To further understand why some different thinking may be required, consider the following plan for a person's use of time:

7:00 - 8:00 Wake up, eat breakfast, shower, etc.
8:00 - 9:00 Drive to work.
9:00 - 12:00 No important business. **Save this time for later.**
12:00 - 13:00 lunch
13:00 - 14:30 Meeting in which new work is assigned.
14:30 - 17:00 Get a jump on the new work by using your saved time from the morning to do a total of 5 and a half hours of work.
17:00 Go home.

Perhaps unfortunately, this kind of scheduling is impossible. An identical problem exists in scheduling the processor, namely that an irreversible decision must be made as to what to do *now*, and processor time may not be inventoried.

All of these difficulties taken together call for a special mechanism for auctioning processor time. Several such mechanisms are outlined in chapter 5. Some of the criteria used to evaluate these models are presented in the following section.

4.3 The Ideal Computation Market

The ideal computation market should allocate the processor efficiently (discussed at length in the next subsection), and also have the following general characteristics:

- Contain an implicit prioritizing mechanism that awards the processor to the most "important" processes, as determined by market forces.
- Serve as a decision-maker by implicitly making rational decisions at each opportunity.[2]
- Contain a simple mechanism to determine bids entered by processes. Also, these bids should have some grounded meaning, and be easy to interpret.
- Allow for dynamic adjustment by making new decisions to accommodate an ever-changing environment.
- Contain a computationally simple mechanism for taking bids and awarding the processor based on these bids.
- Allow a default ordering of processes (a kind of *plan*) to be maintained that can be executed if the costs of decision-making become too great.

[2]As always, "rational" means rational with respect to the agents' preferences. If the agents are random and/or irrational themselves, then the allocations will also appear senseless.

- Execute market calculations very rapidly compared to the time-scale of process execution.

It may not be possible to achieve all of these goals simultaneously, but they will serve as guidance in criticizing possible methods of allocating the processor. The last goal may rely largely on implementation details and available hardware, but is still a very important property for market allocation to have practical applications. Specialized market hardware would be especially beneficial in this regard.

4.3.1 Efficient Allocation

The most important property for a processor market is that it determine an efficient allocation of the processor. Ideally, this would mean maximizing the value added to the system by the production of new goods. I will refer to this as the *ideal allocation*, which I now define formally:

An **ideal allocation** is one which, during any specific period of time, results in the maximum possible amount of value added to the system by the use of computation.

Unfortunately, achieving this goal will usually be impossible, for two primary reasons:

1. In general, achieving this goal will require examining all possible paths of the system. Because each computation may change the value of future computations, there is no other way to determine the maximum possible amount of value that could be generated. However, examining these paths takes time[3] and during this time the processor is not being used in a way congruent with the above definition of ideal allocation. Moreover, in the case where processor allocation will continue into the future indefinitely, it will be impossible to explore all possible paths in any amount of time.

[3] A very large amount of time, in fact, since the number of paths is exponential in the number of processes, and testing each path can require executing every process along it.

2. Even when all of these paths are examined, it may be that the path to creating the maximum value for the first 1 second is not part of the path for creating the maximum value for the first 4 seconds. As a simple example, suppose there are only three processes, none requiring any inputs. Process A can produce something currently valued at $25 in 1 second. Process B can produce something currently valued at $20 in 1 second. Process C can produce something currently valued at $70 in 3 seconds. Further suppose that each process can only execute once (i.e. the value of its service drops to $0 after one execution). Most importantly, suppose that the execution of B causes C to be worth $80, though this is not known in advance. Assume that the execution of A and C do not affect market prices. The ideal allocation for 1 second is to run process A. However, the ideal allocation for 4 seconds is to run process B and then C, since that will total $100, but A then C will only total $95. Thus it is impossible to ensure that during *every* time period the processor has been used for its maximum benefit (in terms of value added).

The realization that ideal allocation is generally impossible motivates the definition of a theoretically realizable goal which I will call an efficient allocation, though it might also be called a rational allocation to draw a parallel with rational decision making. Intuitively, an efficient allocation is one which comes as close to an ideal allocation as is practical, subject to the constraints that future values cannot be predicted with absolute accuracy and that the past cannot be altered.

An **efficient allocation** is one in which the expected future allocation is ideal. Specifically this means that at any instant the process that is running must be the one that is the first step in an ideal allocation for the future given the current valuations of goods (which may take into account speculation concerning future value). An efficient allocation will be ideal if the value of all possible future computations is known in advance.

The notion of efficient allocation, defined in this way, corresponds exactly to what Russell and Wefald [25] [24] call *meta-greedy* algorithms. It corresponds to considering only the set of immediately possible steps, but estimating their ultimate effect in some

time-bounded way. By choosing to limit the meta-decision (deciding how to decide to decide...) to a single level, a very large gain over purely greedy algorithms can be achieved without consideration of the *infinite regress* problem [31] [18] [23].

The so-called meta-greedy approach differs from what is traditionally called the greedy approach in computer science because it can include consideration of future events, in this case in the form of heuristics or deliberation on the part of consumers. For my research, I accept that the time spent evaluating the market (including decision-making time spent by consumers) may not be allocated in a provably optimal way. However, this time can be indirectly considered by including market evaluation time in processes' time estimates, as suggested in chapter 6. Furthermore, several possible approaches to limiting deliberation time are possible, though a discussion of these methods is beyond the scope of this work. The interested reader should see [32], [25], and [26], as well as references cited therein concerning bounded rationality.

The use of the word "efficient" in making this definition also deserves some justification. Efficiency in economics is normally defined in the sense of Pareto Optimality (see section 2.2). Since the defined allocation ensures maximum expected profit from computation, any deviation from this allocation would result in at least one agent being worse off — the agent that is the beneficiary of profits from computation. It is in this sense that the Pareto Optimality condition is satisfied and that the allocation defined above can be called "efficient."

Now let us re-examine the example mentioned in proving the impossibility of ideal allocation, but this time in terms of the definition of an efficient allocation. Since in the example each process produces no value when run again, there is no need to consider more than 5 seconds. The efficient allocation is A then C then B since A maximizes the value added during the first second, C maximizes the value added during the next three seconds, and B maximizes the value added during the final second. The efficient allocation differs from the ideal allocation (B then C for the first four seconds) because the effect of B on C is not known.

If it *is* known that running B will increase C's value from \$70 to \$80, then it is in fact **B** which is creating the extra \$10 in value. In the economy, this will be taken into account by the fact that whatever consumers were willing to pay \$80 for C's improved output will be just as willing to pay the \$10 difference to B – it is of no significance to the consumers who is actually generating the goods they value. In this case the new values added become A: \$25, B: \$30, C: \$70. Then the efficient allocation is B, A, C, which is also an ideal allocation since the future values were known. However, if C will be run many times in the future, every time improved by the fact that B executed, then it is difficult to account for the actual value of executing B. This is precisely the problem of positive externalities, discussed in section 7.6.3.

When the precise results are unknown, but there is some reason to suspect that B may affect the value of C, it is possible that there are speculating consumers and/or consumers who assign utility (and thus implicitly value) to experimentation. Either of these types of consumers may be willing to pay to execute B *just to find out whether B affects the future value of C.* It may be best to consider this service as a separate process (call it D) whose output is "determining if B does something useful." The market will assign some value to that service and if determining B's usefulness becomes sufficiently valuable, then D will execute (and presumably also execute part or all of B).

4.3.2 Comparison to Scheduling

The efficient allocation has one striking difference from what I will herein call the *scheduling problem.* In the scheduling problem, processes are "posted," meaning that they suddenly come into existence and need processor time to be serviced. A scheduler must eventually service all posted processes, a condition known as *non-starvation.* Each process must be executed exactly once; if a process is to be executed repeatedly it must be posted repeatedly. A pure market mechanism suffers from the disadvantage that some processes may never be considered valuable enough to run, in the case where important processes come into existence at a sufficient rate that less important processes are never

serviced.

When determining an efficient allocation of processor time, on the other hand, the available processes are considered as tools. The most applicable tools are used and those that do not become sufficiently useful (i.e. valuable to execute) will not be used. This is exactly what is intended. Just because a process exists does not mean that it performs anything valuable to the consumers, who, as mentioned earlier, are the implicit directors of a computational economy.

If a market-like mechanism is used in a scheduler, then it must somehow be sure that all processes eventually become sufficiently valuable to execute. This is precisely the purpose of Drexler and Miller's escalator algorithm [21], which is discussed briefly in section 5.7.

Although efficient processor allocation does *not* require that tasks become more valuable the longer they have gone unserviced, there will certainly be cases where that will happen. Consider a person trying to allocate her time efficiently. Perhaps an expected commitment turns out to be more important than lunch one day, and later she decides to postpone dinner because she wants to work out before eating. As the "process" of eating is postponed more and more its relative value increases. If by the next morning she *still* hasn't eaten anything, breakfast may have much greater priority than getting to work on time.

Although I have not experimented with it specifically, I am optimistic that the escalator algorithm could somehow be present in the consumer side of the economy, just as in the human example above. Thus the relative valuations of unserviced processes could increase because of an actual change in demand. This would allow the escalator algorithm to be cast into a pure market system, in which bids for processor time directly represented the consumers' relative valuations of goods and services.

4.4 Applications

There are at least two broad classes of applications for a system that economically allocates computation time.

The first class includes any system in which there are many possible processes to execute, but insufficient time to execute all of them. This is the case in the reasoning systems mentioned in the introduction. The football spectator of section 1.1 has several choices, but her time is constrained. If she spends more than a few seconds updating beliefs, she will risk missing the next play or perhaps a replay. Thus a choice must be made about what activity has the greatest value to her. An economic allocation system is well suited to this sort of decision-making.

The second class consists of what I will call *dynamically adjusting algorithms*. The purpose of the economic mechanism here is to be sensitive to the details of the subtasks or to changing structural properties (either in time or encountered during exploration). The market allocation mechanism allows the system to choose the algorithm that is most appropriate to the specific section being dealt with at the moment, and to alter the choice of algorithm as appropriate to the structure of the problem as it is encountered. Research in search by Russell and Wefald [25] has shown that it may be possible to gain significant improvements by specializing the search method used locally. Although that work does not use markets specifically, it points to the possible usefulness of dynamically adjusting algorithms. For the general case of this type of algorithm, the market allocation method should prove very applicable.

Chapter 5

Overview of Possible Models

In this chapter I provide general discussion of several possible models for a market in processor time, including the model that I implemented as part of the Reasoning Economy. Some of these models have been suggested in part by other authors (especially Drexler and Miller [20, 22, 21]), and others are original. Each model provides a mechanism for assigning some type of market value to the good computation. However, the nature of the equilibria that result as well as the specific behavior of the system may be significantly different over different market models. Here I describe the behavior of each market type and discuss advantages and disadvantages of each method.

There are at least three fundamental ways to describe the good which is being sold in a processor auction.

1. The good is a block of time on the processor, defined by a start and end time. For example a process may buy a "license" to compute for 3 seconds from 11:50:25 to 11:50:28. The start time may also be *now* or given relative to some reference time.

2. The good is the processor itself. Control of the system is implicitly transferred to the new owner.

3. The good is the use of the computer. The price refers to the cost per time unit of use.

The decision concerning what good on which to base the market is an important one and has significant implications to the behavior of the system as a whole. Indeed, it can be argued that the choice of good type defines the market protocol and that the other aspects I have used to differentiate models in the following sections are details. However, the distinctions that I have made are intended to exemplify possible behavioral differences in different market models rather than to provide a theoretic taxonomy.

5.1 A Time Block Model

Let us first consider auctioning time blocks which begin *now*, meaning that only the very next block of computation time is sold. This model seems intuitively sensible but presents an immediate problem: The various bidders in the auction are not bidding for the same good. While one process may demand 3 seconds of computation time, another may want 0.5 seconds, and in general it may be that there are as many different goods as there are bidders. For concreteness, suppose that the first process enters a bid of \$6 for a 3 second slot while the second enters a bid of \$4 for a 0.5 second slot. Since the goods bid upon are different, it is difficult to compare these bids and make a decision as to which process actually computes next. One possibility is to consider which process bids more *per second*, and in this case the second process has bid four times as much (\$2/s vs. \$8/s). This line of reasoning can eventually lead to the third approach above in which the computer is auctioned at a per-time-unit price.

Nevertheless, a computation market protocol *can* be based on goods of the time-block nature. However, a shift in perspective concerning what is meant by a bid is required. In effect a process's bid *is* the amount of computation time demanded, rather than a valuation of the good to be purchased. Any number of specific mechanisms can then be built on top of this scheme to decide how to allocate processor time based on these bids. Reasonable possibilities include awarding the processor to the process that has demanded the least computation time (attempting to avoid bogging down the processor with mammoth

tasks) or that has demanded the most computation time (guessing that the long-running tasks will be most important). Other awarding schemes are also possible, though it seems unlikely that a theoretical justification can be provided for any specific method, since the length of time to compute does not carry any inherent description of significance. It is conceivable, for example, that tasks which accomplish nothing (useful) could require any specific computation time. This model, therefore, is not guaranteed to result in an economically efficient allocation of computation time. Thus we are not assured of any the potential benefits of a market discussed in earlier chapters.

Despite the problems just mentioned, for some systems it may be possible to develop a method of awarding the processor based on time-block bids which results in reasonable behavior. The earliest implementation of RECON featured a market mechanism in which the process requesting the most computation time was awarded the processor, provided that the request exceeded a certain minimum time. This method of evaluating bids was purely ad hoc, but was chosen because under the specific constraints of that system, the resultant behavior was not noticeably unreasonable.

A further complication arises in any system in which the precise time of computation of a particular process cannot be computed in advance. This will be the case in the vast majority of systems since precise data is very rarely available for every possible configuration of the parameters and the environment. Thus a process can only determine its exact execution time by executing, leading to an obvious Catch-22. The difficulty then is this: suppose that at the end of the 0.5 seconds that our example process bid for, it has not completed its computation. If the computation does not produce any intermediate values, cutting it off immediately will mean that it has accomplished nothing (it may also be possible to suspend the procedure, which is discussed in chapter 7). This results in a situation often studied in economic game theory. Since there is nothing that can be done to alter the past, the decision is simply whether to allow this process to finish its computation or to terminate it and start another. If the initial time estimate of the currently running process is assumed to be relatively good, it is reasonable to guess that

it will finish soon (the precise time scale is unimportant to this argument). Thus the choice is reduced to whether to (continue to) run a process that will produce a useful result in a very short amount of time, or begin running one that will be starting at the beginning. Under most circumstances, the already running process will be the better bargain.

So what's the problem? Suppose that a few milliseconds later it still hasn't completed its task. The decision to allow it to continue is even more obvious now, because we have every reason to believe that it will be done *even sooner* than when we made the decision to allow it to continue moments earlier. In principle, this allows one process to take over the processor for an arbitrary length of time without any irrational decisions being made along the way.

The skeptical reader may want to consider a real-world example. Suppose you are waiting on the 5th floor and want to get to the first floor as quickly as possible. You know that on average it will take you 2 minutes to get down the stairs, but only one minute to wait for the elevator and ride it down. So you make the rational choice of waiting for the elevator. After 2 minutes of waiting, you are still on the 5th floor, so your choices are to walk down the stairs, which will take 2 minutes, or to continue waiting for the elevator which by now you can reasonably expect will take well less than one minute. This process can of course continue, but even if the elevator does come immediately, it still took you longer than if you had gone down the stairs. In retrospect it appears that you made the wrong decision, but in fact you made the best possible decision at every opportunity.

This problem for a particular market mechanism is one which must be addressed, but which has no general solution. It is conceivable to allow controlling processes to compute indefinitely or to terminate them when they have exceeded their bidded time slot by a specified amount. The second method is likely to be the best, but it introduces a certain element of risk for any process, which must somehow be accounted for in bidding. The risk arises because the process may be terminated before it has achieved any result valued

by the market. There is no reason to believe that this element of risk is a flaw, but it is important to recognize its existence because it complicates the bidding strategies required for processes. I have not investigated these strategies in any detail, however, because of the significant disadvantages of this model of computation markets outlined above.

5.2 A Futures Market in Time Blocks

A second model in which the processor is still auctioned in blocks of time is suggested by real world futures markets. In this model, the rights to future time slots are sold in advance of the actual computation opportunity. Many of the same problems discussed in the previous section apply to determining exactly how to sell these slots. Again the goods demanded are non-uniform, making it difficult to evaluate bids. It is conceivable to take bids in length of time desired, as in the previous time block model, or to break slots down into some smallest length, and allow purchasing of contiguous slots to perform longer calculations. It is also possible to create a sort of default queue by simply giving each process, as it is encountered, the next available slot in the future. A somewhat different possibility is discussed in the next section. In general, the details of how to sell the future time slots could be very complicated, but significant problems arise in this model regardless of the method of initial sale, so I will not discuss these details.

Instead, let us assume that the rights to all relevant future slots have already been assigned, so that a sort of queue has been formed in which each process owns a particular window in which to compute. It is conceivable not to use the market mechanism again and to simply allow each process to compute in the order determined. This gives up some of the advantages of using markets, in particular capabilities for dynamic adjustment. The system has only used economics as a planning tool and now executes that plan. Since the market mechanism itself cannot easily predict the results of each step of the plan, this model of planning is likely to be significantly inferior to more traditional planning methods in computer science. However, the use of economics in planning problems

probably deserves more exploration.

As an alternative, processes may sell their time slots (or any portion thereof) at any time before the slot to be used has passed. This allows the market to adjust dynamically to results of past computation. This complicates the coding of processes, however, because they have to have mechanisms and methodologies for buying and selling time slots from each other, which in general is more complicated than selling computation time in a central auction. At this stage we again encounter the problem that different processes may be trying to buy different amounts of computation time from the same process.[1]

Unfortunately, the futures market model has a much more significant problem. The initial queuing of procedures places arbitrary boundaries and divides up the use of future time into specific slices. However, there is no reason to believe that this slicing will be the correct one for all possible paths of the system. This places large, inhibitory constraints on making adjustments to future needs. Consider the following initial awarding of future time slots:

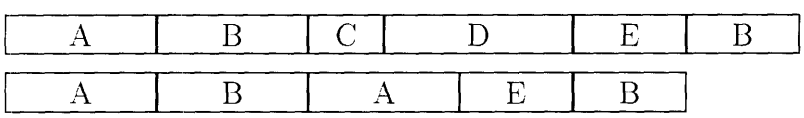
Further suppose that the system begins executing these time slots and the following ensues: Process A produces its result without any significant affect on market prices. Process B produces a result that greatly increases the market value of a result that can be produced by process A. So process A wishes to buy time slots from C and D. It can buy all of C's time slot for whatever price can be mutually agreed upon. However, it only wants to buy a small portion of D's time slot so that it can complete its computation. Assuming that D can produce no useful results if left with a smaller window, D is unwilling to sell unless it can sell its entire time slot. So either A must buy the entire time slot (in which case the latter part may go unused[2]), or D must find another buyer(s) for exactly the remaining time slot. Such a buyer is often not available, though it may

[1] Anyone who has played the card game PIT is aware of how difficult it can be agree to trades when different buyers want different, but overlapping, parts of what you have to sell.

[2] Because computation by A may change market values of other executions, it is also possible that A could sell part or all of it unused time *after* executing.

be advantageous to design the system with one or more "omnipresent" processes that automatically execute during dead time created in this way. The result is that large blocks of time may be either misallocated, as when the process using the otherwise idle time is not actually the most valuable process at that moment, or unused, which is likely to be an inefficient use of resources. (There are several cases where idle time may not be inefficient. Two representative cases are: 1) An event is expected to occur in exactly 2 seconds which needs immediate attention when it does occur. A process may be willing to bid for the action of waiting two seconds and then attending to the event immediately. This option will get more and more attractive as the time "wasted" to ensure immediate service gets smaller. 2) The parallel architecture situation discussed in section 7.7 in which a process must hold some processors idle in order to obtain enough processors to perform its desired computation.)

A third alternative is to require A to buy out D completely (at whatever price is mutually agreed upon), and then move up the rest of the queue since A will not be using the entire slot. This outcome is perhaps clearer diagrammatically. The original ownership is shown directly above the new ownership of future time slots.



In the general case, this adjustment may not be valid because a process may have purchased a time slot beginning at a certain time for a specific reason – perhaps because of an event expected to occur at that time. Such a process cannot simply be moved up in the schedule because it may have nothing valuable to do at the earlier start time.

When this type of adjustment is allowed, the futures model degenerates into the time block model discussed in the previous section. What is happening in reality is that at each opportunity, any process can buy the next time slice (from the current owner(s)). The initial assignment of owners to future time slots only serves to complicate matters and to require more elaborate market mechanisms for processes to buy and sell from each other. There is, however, a small advantage to structuring this model as a futures market,

even though the behavior should be identical (except that the market calculations will be slower when the queuing is revised regularly and faster when the queuing remains unchanged). This way of designing the market allows the user to see a sort of planning and replanning of tasks. Since there is always a queue of future tasks, there is a sort of implicit statement of what the system "plans" to do if nothing along the way makes it necessary to rethink this plan. This is also may prove useful if the cost of replanning becomes too great for some reason. In some situations it is better to act then to spend any time considering what to do next.

5.3 Time Blocks Within a Window

Another way of auctioning time blocks is to require that each process specify how long it wishes to compute and supply a time period in which the computation must occur. In addition it should supply the market value of its computation. For example, a process might be able to produce $50 by using 3 seconds of computation beginning any time between 11:51:23 and 12:02:15. It is not obvious how these bids would be determined, but let us assume that all processes supply the auctioneer with bids of this form. In many cases, it may be possible to satisfy all of the bidding processes – algorithms exist to determine such a solution if one exists.[3]

In the case where not all processes can be satisfied, the best solution is to remove the minimum total value of computations that will allow all remaining processes to be satisfied. Although this is certainly possible, the problem is clearly NP-Complete because it require repeatedly solving the "Sequencing within Intervals" problem of the previous paragraph.

At this point, we have assigned some sort of ownership to future time slots. Therefore, further analysis of this model is identical to the futures market presented in the previous section.

[3]However, the problem is NP-Complete. See [15], pg. 70, "Sequencing within Intervals."

5.4 Transfer of Processor Ownership

If the auctioned good is considered to be the processor itself, it can be bought and sold as though it were any other good, with the current owner having the implicit right to compute. The structure of this market is appealingly simple. Processes enter bids for the processor and if the current owner wishes to sell the processor to the highest bidder then it does so. In a one-processor computer, the entering of bids cannot take place continuously, so presumably an auction would be held by the owner whenever it wished to consider selling the computer. An optional rule could also be included that the seller cannot decide to keep the computer after the auction, but *must* sell it to the highest bidder. This sort of rule is usually used in auctions involving human buyers to preserve a sort of "fairness." This kind of fairness, is not, however, demanded by computational processes.

Despite its simplicity and apparent elegance, auctioning the computer itself has several severe difficulties.

The first problem arises on the bidding side of the auction. If we assume that each process can determine a valuation of the computer based on the market value of its execution, we would expect it to enter that value as its bid. For concreteness, let us suppose that there are three processes and label them A, B, and C. Process A values the computer at $100, B at $50, and C at $15. However these bids reflect only the fact that each process will use computation time to perform services or computations that have market value. For example, when process A finishes computing it can expect to be paid $100 for its services, since that bid was determined from the market value of executing. However, process A still owns the computer and can now sell it. The computer has not devalued in any significant way (as would have a car, for example), so A can now sell the processor for $50, the bid that B entered in the last auction. The result of A may alter the exact valuation by B either up or down, but since this cannot normally be anticipated, $50 is likely to be the best guess.

Since all of this can be calculated at the time of the initial auction, process A should

change it's valuation to $150. This reflects the true value of the processor to process A. Similarly, if process B is awarded the computer in this auction, it can expect to sell the processor to A for $100 after performing its own computation. Thus process B adjusts it valuation to $150 as well. On this iteration, process C will change it's bid to $115, expecting to be able to sell to A upon completion.

In the equilibrium, all three processes will enter bids of $165, each expecting to sell it to one of the others who will in turn sell it to the third. Unfortunately, this *is* the general case of this type of processor auction. If each bidder enters its true valuation of owning the processor, all bidder will enter the same value. This value reflects only the system's *total* valuation of the processor and the auction therefore does not provide any mechanism for determining which process should execute next. The end result is that no economically efficient allocation of processor time can be determined.

Of course, in a computer system, processes can be programmed so that they do not attempt to examine (or even predict) each other's bids and bid instead exactly at their individual valuation. This will result in extra profits from the sale of the computer after completing execution, but some mechanism for redistributing these profits could be developed. The larger disadvantage of this method is that it will preclude most learning algorithms for improving the quality of bids over time, since those methods would be expected to converge onto the true valuation (as above).

A second problem arises from allowing ownership of the resource that controls the entire system. Once a process has purchased the computer, it can, at least in principle, complete as many computations as it desires. If it turns out that it values performing an additional computation more that it values money (with which to perform future computations), it may continue to compute, optimizing its own utility at the possible expense of the system as a whole.

However, this problem can similarly be avoided by programming all processes so that they surrender the computer when they have completed the task for which they entered the original bid.

The final problem is much more fundamental, and refers to the nature of the equilibrium determined by this type of market. Because processes are bidding for *possession* of the computer, the value bid reflects only the market value of what it can eventually accomplish, with no accounting for how long it may take. For example, process A can perform a service that is valued at $100: determining if an input program will halt in less than 1 hour. Process B can perform a service valued at $10: determining the length of an input program. Suppose further that *every* execution of process B is worth $10, though this would often not be the case. Clearly, the $100 process will execute first, but in general it may take 1 hour for it to determine its result. During this same time, process B could have determined the length of an enormous number of input programs. Since each service remains constant at $10 (by assumption), the processor could have been used to generate orders of magnitude more than $100 worth of results during the hour that it was used. Since all of this information was known in advance, this market mechanism has misallocated the computation resource.

The crux of the problem is on the bidding side of the model, and is an extension of the bidding problem discussed earlier in this section. In the above example, B has undervalued the computer because it has no way of determining how long it should own the processor. Since no process needs to own the computer for its entire lifetime, it is difficult if not impossible to determine how much to be willing to bid. In the over-simplified example above it turns out that B should buy the computer and then compute indefinitely. But there is no systematic way of determining how much to bid for ownership. Since it expects to profit indefinitely, perhaps it should bid an infinite amount for the processor! This does not seem to make very much sense. To understand this problem further, imagine that you are immortal and are entering into a market to buy a house that will never need any maintenance. How can any specific maximum utility be assigned to this? Of course, a maximum *bid* can still be determined by bidding all remaining capital (plus loans if available) when other "needs" have been handled.

If the processor is made to devalue with time and the processes are made mortal

(meaning that they are permanently removed after some maximum amount of time), then these problems are greatly alleviated. However, since there is still no way to determine in advance how long a process will want to own the processor, there is no systematic way of assigning value to ownership.[4]

For all of these reasons, a model based on the transfer of ownership of the processor cannot be guaranteed to result in an efficient allocation of processor time.

Sale of the processor can also be extended to a futures market which resembles real-world futures markets even a bit more than the time block futures market described earlier. This variation would allow processes to make an agreement that one would sell the processor to the other at a specific time for a specified price. This represents a sort of speculation on the part of the processes concerning the future value of computation time.

The extension to a processor futures market is not a distinct model because the owner of the processor is, in general, free to make whatever sorts of sales agreements it wishes. The disadvantage of allowing speculation is that the allocation of the resource is committed to in advance and may not be the best allocation possible when the time comes. However, the futures market does offer one of the principal advantages of its real world analogue, namely the ability to transfer risks. A process buying the computer may be taking a certain risk that the future value of the computer will be significantly less than expected (or even zero, as in the case where the process "completes" the task defined by the consumers' demand). In a futures market, a process may eliminate risk by agreeing to a mutually acceptable future price at which it knows it will be able to sell the computer. In a case where the process would not have been willing to bid given the risks involved, the futures mechanism may prove very useful.

[4]When there are a large number of processors available, such that every process can eventually expect to own a processor, this problem may disappear. The market becomes very much like a real-world housing market. This is in essence a matching problem. See [19] for a thorough analysis of a closely related problem.

5.5 Processor Rental

In this section I introduce a model based on renting the processor at a rate per unit time. This corresponds to the third good description at the very beginning of this chapter. Because a form of this model proves to be successful in assuring efficient allocation of computation time, I have implemented a first-generation version of this model within the Reasoning Economy (RECON). A description of the implementation and further comments about my experience in practice can be found in chapter 6. In this chapter I describe rental models abstractly and show that the "improved' form will result in an efficient allocation.

5.5.1 The Pay Phone Analogy

The simplest model of renting the processor is suggested by an ordinary pay phone. In this model customers (processes) pay according to some fixed rate scale for the use of the phone (computer). When more than one process desires to use the processor at the same time, they form a first-in first-out queue. Thus this system corresponds to queuing up processes which are willing to pay at the set rate. This system does not place any priority on processes according to how much they are willing to pay, however, and thus defeats the primary purpose of using a market. In addition, when no processes can afford the going rate, the processor will sit idle when perhaps many useful computations could be performed.

5.5.2 Improved Processor Rental

A simple modification to the pay phone model results in an enormous improvement. For the remainder of the thesis, the "Processor Rental Model" will refer to the following extension: The price is now allowed to be fully variable and the processor is awarded to the process that is willing to pay at the highest rate. Only if no process enters a non-negative bid will the processor remain idle. If idle time is especially undesirable, a

set of processes that are always willing to compute at a price of $0 may be made part of the system. However, in some systems (including RECON) it may be desirable to discontinue computation when the processor reaches a value of $0 per time unit. This will often correspond to the situation where all of the consumers have been satisfied or removed, implying that all "goals" of the system have been achieved.

The auction side of this model is straightforward. Given all of the bids the processor is simply awarded to the process entering the highest bid. The exact rate that is charged can be anything greater than the second-highest bid and less than or equal to the winner's bid. (See [21], section 2.1.1 for concise discussion of relevant auction mechanisms.) The winning process is then free to compute at that rate until it runs out of money, completes its desired computation, or is outbid by another process. The last may be barred by conducting a new processor auction only when a process releases control. The issues of interruption and suspension (and a related constraint which I have called atomization) are discussed in chapter 7.

To determine processor bids, we must find the maximum rate at which the process can expect to make a profit by computing. If the process expects to have a loss incurred by computation, then it will not be willing to compute. Thus the maximum rate that a process will accept will be its expected profit from computation divided by some estimate of the time that it will take to perform the computation. Processes which wish to guard against any possible losses may estimate their execution time conservatively (meaning on the high side). Specifically, the maximum bid rate is given by:

$$maximum\ rate = \frac{value\ of\ outputs\ and\ services - value\ of\ all\ other\ inputs}{estimated\ execution\ time}$$

At the bid rate determined by this formula, a process expects a net profit of exactly $0, since the total amount paid for computation will be equal to the value created by the computation. At any price below that rate, the process will expect to make a profit, so will be willing to compute. Symmetrically, at any price above that rate, the process will

expect a loss, so will not be willing to compute. Note that a process usually takes some risk because it cannot know with absolute certainty that it will be able to complete its computation in any specific amount of time. In some situations it may also be possible that the market value of the outputs declines during computation. This particular risk can be avoided by agreeing to the sale price of outputs before beginning computation. Over the long term, a process that executes repeatedly and has a reasonable ability to estimate its execution time, should expect these risks to be minimal since accidental losses (from a low time estimate) will be compensated for by accidental profits (from a high time estimate).

When a process can produce different total values of results in different amounts of time, it is not quite as clear how to evaluate the maximum bid formula, since some of the terms can have many values. However, what we are interesting in is the *maximum* rate at which the process will want to compute for *all* combinations of parameters. As long as a process can store all of its valuable results, we are guaranteed monotonicity of output value over time. Thus for an arbitrary function of output value vs. time spent computing, the maximum bid rate corresponds to the slope of the steepest line through the origin which touches the curve at some point. Figure 5-1 shows a canonical case where all valuable results are generated at the end of computation and a more general case with a procedure whose result improve given more time to compute. In each case the slope of the dotted line indicates the maximum bid rate. The intersection point indicates the ideal amount of time for the procedure to compute since at that point it will have generated value at the greatest possible rate. In the future any process may bid again based on the value that it can continue to create, though at a slower rate.

The numerator of the maximum bid rate fraction represents the value added to the system by the use of computation time. Thus the bid itself represents the average rate at which value is added by computation. This observation enables an informal proof that this model will indeed assure an efficient allocation of processor time. (Refer to section 4.3.1 for the definition of efficient allocation.)

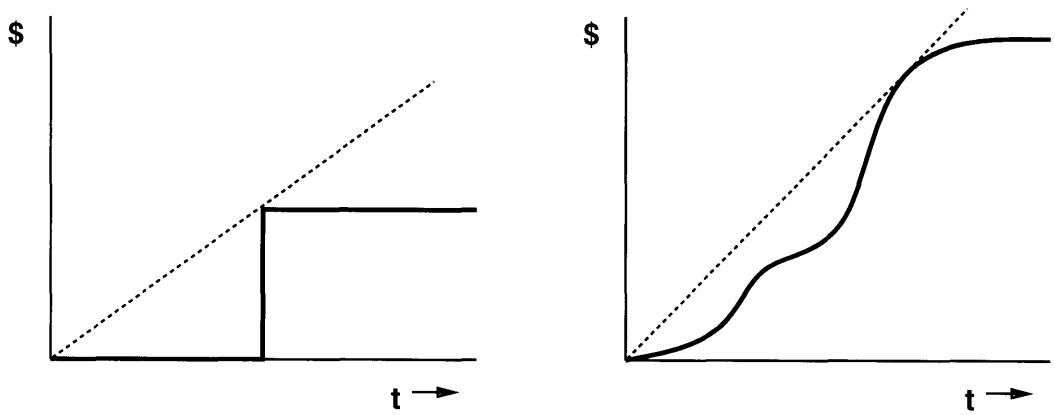


Figure 5-1: Graphical Maximum Bid Rate Determination.

The model awards the processor to the highest bidder and thus to the process generating value of the greatest rate. Since at each instant the rate of value being added is maximized, the value added during any expected future time period (equal to $\int_{t_0}^{t_1} b(t)dt$, where $b(t)$ is the bid rate at any instant in time t, and t_0 is *now*) is also maximized. Therefore, this model *does* achieve an efficient allocation of processor time.

There are several important assumptions implicit in the argument above. Firstly, an action which is in fact discrete has been made continuous. Specifically, the actual value of a computation will often not be generated until the end of execution (in fact it may be desirable to *require* this kind of "atomization," discussed in chapter 7). Thus any process which is interrupted or runs over its time estimate will not actually be generating value at the desired rate.[5] Even without this problem, equating the value added with the integral above is not precisely correct. But whenever t_1 corresponds to the end time of some execution, the integral will in fact be equal to the total value generated during that time.

However, if any process must be terminated before it is complete, then it is possible that more value could have been generated by running one which would have finished.

[5]The case of running over the time estimate is the same problem discussed in section 5.1 with the elevator example. Unfortunately, this problem cannot be avoided, and, as before, it will nearly always be best to allow a process to continue to run. However, when time estimates are reasonably accurate, the total errors should balance out over the long run.

Thus the general case in which all computation must cease at a predetermined moment deserves more attention.

The situation is similar to the age-old hypothetical question posed to humans: "Suppose you had exactly two weeks to live. What would you do?" Being placed in this situation may result in rather different choices both for the person and for the computer system. A complete discussion of this specific case is beyond the scope of this work, but I will make several comments. The simplest improvement that can be made is to simply disallow bidding by any process whose time estimate exceeds the amount of time remaining. This will reduce the chance of the "last" process being terminated before it completes any valuable computation. Further improvements under a predetermined time constraint are very difficult because it not known exactly how each action will affect the value of future actions. Thus no effort to fill up all of the allotted time in advance can be guaranteed to succeed.

When these caveats are understood and accounted for, the processor rental model detailed in this section can be ensured to provide efficient allocation of the processor and rational use of time.

5.6 A Combination Model

It is also possible to combine various methods of selling the processor to be used at different times in different situations. For example the processor might be auctioned in time blocks, but the owners be allowed to rent parts of their block to other processes. Almost any combination is theoretically possible; the analysis is usually a simple combination of the analyses of the component models. I discuss here only the model suggested by real-world markets: Processes participate in an auction to buy the processor and then the owner may rent it exactly as in the rental model of the previous section.

This model carries all of the same bidding difficulties as the standard processor sale model. The advantage offered by this combination is that when owners are assumed

to be profit maximizers, the guarantee of efficient allocation is carried over from the rental model. Indeed, the owner(s) in the rental model were abstract – only their profit-maximizing nature was required for the analysis presented there. The addition of processes as owners, though still just as difficult to administrate, may give a sense of the general goal being attempted by the system. Though I have demonstrated that there is no general solution for processor auctioning, it may be possible in some systems to achieve a reasonable approximation. If this is this case, it may be possible to learn something about the system's behavior by examining the chain of ownership.

5.7 Drexler and Miller's Escalator Algorithm

Drexler and Miller outline a model for auctioning the processor in [21], which they call the escalator algorithm. This model is intended for use in what I have herein called the scheduling problem. It is designed to allow some prioritizing while still ensuring that every posted process eventually executes. In this section I briefly describe this model and explain why it cannot be carried over into the domain of the efficient allocation problem discussed in this thesis. It should be emphasized that this is not in any way a refutation of Drexler and Miller's work.

The escalator algorithm transfers control of the processor to the highest bidder just as in the processor sale or rental models described earlier. Once a process gains control of the processor, it is fully serviced, meaning that it completes any desired computations. The bidding side of the market is completely different, however. Each process is placed on an "escalator," meaning that its bid will automatically increase linearly with time. There are two additional market parameters that specify the maximum initial value of any bid and the maximum rate at which a bid can "escalate." Drexler and Miller demonstrate that under these conditions, any process which is placed onto the fast escalator allowed must eventually execute. Processes not on the fastest escalator will also eventually execute except under very unusual circumstances.

The initial bid value and the choice of escalation rate define in some way the urgency of a particular task. A maximally urgent task will enter the maximum allowed initial bid with the maximum allowed escalation rate. Other tasks will choose lesser values. However, these values are not determined by market forces. The values of the bids in fact carry no specific meaning as to the value of the task. Increases in bid value are automatic, and also may not reflect any actual increase in the value of performing that task.

Because non-starvation is unimportant in the efficient allocation problem, these bids do not carry the desired meaning. Specifically, there is no information available as to the expected value added by a particular computation, so it will be impossible to achieve an economically efficient allocation as defined herein.

Chapter 6

Implementation

I have implemented a very simple form of a processor market within the Reasoning Economy [12]. In this chapter I explain the implementation and comment about its observed behavior.

6.1 The Underlying Architecture

In this section I describe the system into which my processor market fits from a top-down perspective. These divisions are not absolute in the code, but the system is roughly divided into three packages, not including the WALRAS system on which the reasoning economy is built.

6.1.1 AMORD

AMORD [3] is a pattern-matching rule-based language. AMORD keeps a record of possible beliefs, each labeled as "in" (believed) or "out" (not yet believed). The Reason Maintenance System (RMS), described in the next section, keeps track of these labelings. AMORD then queues up facts and rules from the "in" group based on matching the pattern of rules with believed facts. The AMORD system itself, in the current implementation, does not use any economic principles in guiding its operation. Specifically, facts

and rules to combine are not queued up according to any sort of prioritizing mechanism.

The execution of AMORD results in new reasons for beliefs, which may call for updating of beliefs. See section 1.1 for general discussion and an example regarding belief maintenance.

6.1.2 RMS

The Reason Maintenance System is responsible for all of the updates which may be mandated by the operation of AMORD. Among its many tasks is the organization of beliefs into *nodes* (containing a single fact or rule) and *locales* (groups of nodes, not necessarily forming a partition, which are somehow related to one another). The RMS permits reasoning in one or more locales, but the current implementation of AMORD conducts all of its reasoning in a single locale.

The RMS must also relabel nodes from "in" to "out" or vice-versa as necessitated by the reasoning being performed. (For more discussion of reason maintenance and nonmonotonic reasoning see [6], [7], [8], [9], [11], and [10].) The RMS uses the Reasoning Economy (RECON) to determine which relabelings to perform when.

6.1.3 RECON

As mentioned in section 3.4, the Reasoning Economy is built on top of Wellman's WALRAS market-oriented programming environment. RECON creates a good for each possible service that can be performed by an available process. It also creates specialized consumer(s) that assign utility to the particular service. The two most common services requiring computation as an input are node relabeling and locale relabeling. A node-labeler examines a particular node and (re)labels it as "in" or "out," as directed by the RMS. The locale labeler performs a coherency update on all of the nodes in a given locale.

From the utility functions and endowments of currently active consumers in RECON, WALRAS determines the market prices of all of the goods and services. Most important

among these for the processor market is the market value of each of the possible actions: the various node-labelers and the locale-labelers.

The processes which perform these services are instantiated as *producers*. As such, they must provide bids for all of their required inputs, including computation. The WALRAS environment provides generalized procedures for bidding, so that all that is required is to create new methods specialized to computation.

6.2 The Processor Market

In order to implement the processor market within the RECON system, several simple extensions were required. First a good representing the rental of computation had to be established. I called this good COMPUTER-CONTROL since the ownership of a single unit entitles the owner to compute (at the market rental price per unit time). The auction for COMPUTER-CONTROL is established directly by WALRAS, but supply and demand bids must still be defined.

Since the processor is a unique resource in the RECON single-processor system, the supply of COMPUTER-CONTROL in each auction should be exactly one. Since the supplier of COMPUTER-CONTROL must be a profit-maximizer in order for the rental model to ensure an efficient allocation, a specialized producer named COMPUTER-SUPPLIER was created. The COMPUTER-SUPPLIER really just represents the processor itself, but as a WALRAS agent it has a production function that enables it to "produce" one unit of COMPUTER-CONTROL with no inputs.

With these agents a WALRAS auction can be created for COMPUTER-CONTROL that will automatically select the highest bidder in equilibrium. All that remains is to construct a bidding method for producers which require computation an input. Producers of this type are all derived from a subclass of producers called COMPUTATION-PRODUCERS. Therefore the DEMAND-MAPPING method specialized to COMPUTATION-PRODUCERS simply implements the maximum bid rate formula of section 5.5.2.

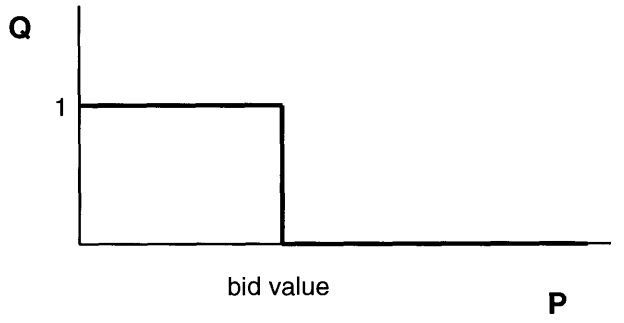


Figure 6-1: Demand Curve for Individual Producer.

The DEMAND-MAPPING returns a procedure which takes as input a price and outputs the number of units demanded by the producer at that price. For the single-processor environment, this output is simply 0 when the price is above the maximum bid rate and 1 when the price is below the maximum bid rate.

In order to create this mapping, each producer must have an associated time estimate, which I have implemented in the form of a method called TIME-ESTIMATE. A TIME-ESTIMATE method takes a producer as its argument (and thus indirect access to most environment variables) and computes an estimate of the number of seconds that will be necessary to perform the desired service. For the RECON environment, these estimates are based on statistical data from several runs of a blocks world reasoning example. It is uncertain if these estimates would generalize to other domains.

In addition, I have chosen to include in each time estimate the amount of time that finding the market equilibrium will require. This gives proper economic incentive to run procedures that accomplish relatively larger tasks and will not require "wasting" time by repeated running of the marking equilibrium algorithm.

This DEMAND-MAPPING implicitly created a complete demand "curve" which is a step function, shown in figure 6-1.

WALRAS then aggregates all of the demand for COMPUTER-CONTROL and finds the equilibrium price as the intersection of the aggregate demand with the (aggregate) supply, shown in figure 6-2.

This mechanism determines the auction winner and the equilibrium price of compu-

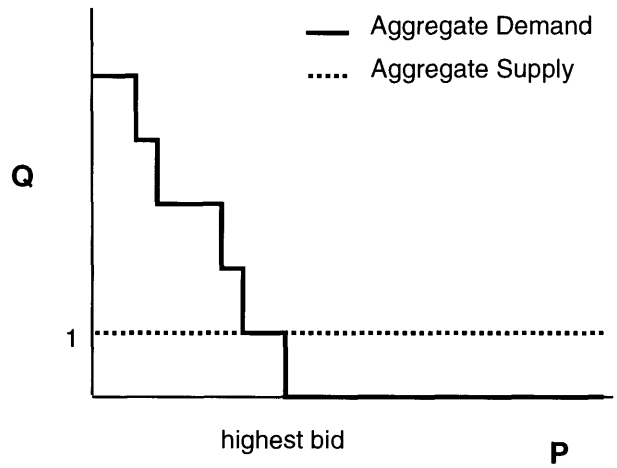


Figure 6-2: Aggregate Supply and Demand Intersection.

tation. Note, however, that the equilibrium is not usually unique, as the curves overlap each other on a horizontal segment. At any price along this segment (open on the left end and closed on the right end) this market is in equilibrium. The current implementation does not take this account specifically and simply uses whatever price the iterative convergence method employed by WALRAS finds. However, the auction with the most justifiable behavior (see [21]) seems to be the second-price auction, in which the price charged to the winner is the minimum equilibrium price.

In the simple RECON environment, the processor market is really only used to choose between the option of relabeling a single node and relabeling the entire locale. However, the results are encouraging, as the system seems to perform as would be hoped at an intuitive level. When the size of the locale is small or when a very high percentage of nodes need relabeling, the market tends to choose the locale labeler since its execution has great benefit and the time to execute will be acceptable in either of these two cases. However, as more and more beliefs become part of the system, the individual node labelers are selected much more frequently because the benefit of total coherence is outweighed by the large amount of time required to ensure it.

Chapter 7

Further Issues and Open Problems

The development of computational markets is really in its infancy. In this chapter I discuss some of the major issues that remain as well as several of the possible difficulties in using the computational market in practical applications in the future.

7.1 Consumer Preferences and Bidding Methods

The research and models presented in this thesis are limited in rationality in that they make rational allocations based on the consumer's preferences. If the consumers in the economy make "irrational" judgments about which goods and services are preferred over others, then the system as a whole will seem to exhibit irrational behavior, even though the processor allocation mechanism is still performing rationally within its domain.

Thus the task of the programmer developing within a market environment consists not only of programming the actual routines which will be the producers in the economy (see section 7.4), but in specifying consumers by their utility functions and initial endowments of goods, including the monetary unit. A few general guidelines are suggested directly by my research:

- Consumers that are endowed with more total value of goods will have a greater influence on the allocations of the processor, at least in the short run. This principle

follows directly from the fact that these consumers will be able to bid more for the goods that they value than those with smaller endowments.

- In the long run, consumers which make the most profits will have the greatest influence on processor allocation. In general, consumers profit from some sort of ownership of producers. However, the details of profit (or loss) distribution also constitutes an open problem. It appears that a well designed transaction and distribution mechanism could serve as a kind of consumer selection device. Those that profit over the long run will be the principal directors of the system in the future, while the others will be bankrupt or represent relatively insignificant demands. Of course, selection does not solve the aforementioned problem of "irrational" consumers.

- Just as in economics, utility functions need only give relative preferences. The specific values of the utility for a particular consumer and bundle of goods are of no significance except relative to one another.

Unfortunately, these guidelines only provide a rudimentary foundation for building consumers. In general several very large problems remain:

The number of agents. At an abstract level, the number of agents participating in the economy will determine how decentralized the decision making process really is. If there is only one consumer the processor will be allocated solely based on that consumer's demands. If there are many consumers with comparable wealth, no individual will have a strong influence on the market prices. In some cases increasing the number of agents may simplify the task of individual agents, as is the case with arbitrageur agents (see [30], section 4.3). It seems likely that there is no way to determine in general the "optimal" number of agents.

Utility functions. The consumers' utility functions will determine what the system considers to be important or unimportant. Presumably these utility functions should

somehow be compatible with the preferences of the user of the system. Otherwise the computer will spend time producing results that are not significant to the user's application. However, establishing realistic utility functions for humans has proven difficult in some cases. It remains to be seen if the construction of these functions is easier or more difficult for computational economies.

A further problem arises if new goods or services may be invented by the system while it is running. The programmer may not be able to anticipate all such goods and thus the consumers may be unprepared to establish preferences concerning tradeoffs between the old goods and the new ones. In the RECON system, this problem is dealt with by introducing new consumers which value the new good and bring endowments that include at least some of the "old" goods. This enables tradeoffs to be established between all goods without requiring existing agents to place any utility on the new good. This method appears to be acceptable within the RECON domain, but seems somewhat arbitrary and unsatisfying. Certainly human agents establish utilities (implicitly) for new goods regularly.

Endowments. As mentioned above, consumers with larger endowments will have greater influence in the economy. Thus the decision concerning the endowment for each consumer is a non-trivial one. One possible approach is to simply endow every consumer equally, but this cannot be justified based on "fairness," since fairness is not necessary for computational agents. Again it seems unlikely that a theoretical justification can be provided for any particular scheme of determining initial endowments.

Grants. In a computational economy, it is conceivable to create money and grant it to particular consumers. I have not explored the desirability of such an option or how it would affect the economy as a whole. Obviously, it would have macroeconomic implications such as inflation, and perhaps is best studied by applying research from macroeconomics.

Bankruptcy. Since consumers will spend their money primarily on services, those that do not own shares in any producers may eventually become bankrupt. In this case it is either possible to remove the consumer or to provide the consumer with a grant so that it may return to bidding. Although the latter option could result in an incentive to go bankrupt, a computational economy can be controlled more directly and consumers can be made to operate without considering impending grants. In general, though, it seems more sensible to remove these consumers or to try to avoid them entirely be having profits from computation redistributed to all consumers.

As mentioned throughout the earlier chapters, the consumers will serve as the implicit directors of computation in the computational economy. As a result design of consumer agents is critical to the success of a system based on a computational economy. Although this may seem like an enormous restriction on the usefulness of this model, it is actually a natural consequence of rational decision-making. In order for any reasoner to make a decision about what is important to spend time on, it *must* have some way of determining the relative importance of difference possibilities. This is precisely what preferences are. Though this does not prove that consumers and computational economies are required, it indicates that any system which must make allocation decisions must have some sense of preferences. Thus to a certain extent the above considerations and problems may be unavoidable.

On the other hand, it is not clear that consumers must be designed perfectly. In some cases it will obvious that one service should be preferred to another. For the muddled cases, it may not be terribly important that the preferences be exactly correct. The system may choose to do something that isn't necessarily the "best" possible option for the task at hand, but it's likely to be much closer than if it had been attempted with *no* preference information. Only further experimentation with computational markets will reveal how important the details of consumer design really are and to what extent it will be possible to develop user-friendly environments for creating them efficiently.

7.2 Markets for Other Computational Resources

Although my research is concerned primarily with allocating the processor, there is no reason that market allocation mechanisms should be limited in this way. Indeed, many of the same benefits of economics espoused in chapter 2 and elsewhere would be desirable for allocating such resources as memory space and disk space. It is even conceivable that resources such as windows or regions of the physical screen be allocated in a market.

There is usually still a significant time dependency, since, for example, memory might be used during a particular computation and then freed. As a result the rental protocol again seems to be the most attractive. There are many unresolved details remaining, but combining some of Drexler and Miller's research with bidding methods parallel to those suggested for processor markets yields a general framework for rental of other computational resources. Here I use memory space as the canonical example and do not address the possible subtle differences between this market and other similar resources mentioned above.

Drexler and Miller have detailed a market mechanism for economic allocation of memory space[21]. This model is general enough to be applicable in the domain of this thesis. Design of agents and auction rules for this system are detailed and carefully analyzed in [21]. Careful attention is also paid to such considerations as allocations of blocks being generally more efficient than allocation of individual locations. Here, however, I ignore these details and present only the salient aspects of their model. I have also adapted and extended certain parts of the theory to include market values (see section 4.1).

The so-called rental-auction algorithm consists of processes entering bids for memory space on a per unit memory per unit time basis. Translated into the general equilibrium framework used to "solve" markets within my research, this would mean producers giving a demand (in memory units) at any given price (per unit memory per unit time). Memory space can then be priced in the market, and rented to those processes desiring memory at the market price. In Drexler and Miller's model, bidders must also supply a price at

which to drop the memory. However, in this framework, the "drop" price will be inferred from the point at which the demand is zero.

Because they do not have a market for all goods and services (section 4.1), Drexler and Miller do not describe a specific method by which processes can determine how much they are willing to pay for memory storage. However, the bidding analysis of section 5.5.2 can also be applied here. A process can determine the maximum rate at which it would be willing to rent memory by the following formula:

$$maximum\ rate = \frac{value\ of\ outputs\ and\ services - value\ of\ all\ other\ inputs}{(estimated\ rental\ time)(estimated\ memory\ units\ required)}$$

This formula reflects the fact that at any price above that rate the process would expect to lose money as a result of computation, since the total expected rental on the memory locations will be the bid rate (above) times the number of locations rented times the time that the memory is used. The resulting expected total cost of memory usage will therefore equal the expected profit when the price is exactly the maximum rate above. At any price above the maximum bid rate, the cost of renting memory would exceed the profit from its use, resulting in a net loss. Symmetrically, renting memory at any price below that level would result in a net profit.

At a quick glance it appears that the formula above could be combined with the maximum bid rate formula for processor time (section 5.5.2), to show that the bidder will always overspend. However, this is not the case. The phrase "all other inputs" used in both formulas includes the other. The maximum bid rate for processor time must take into account the cost of memory as an input, while the maximum bid rate for memory space must take into account the cost of computation as an input. This interdependency is common in economics and does not interfere with determining the equilibrium by iterative convergence.

In fact, the interdependence reflects the very important tradeoff between memory usage and time of execution, and allows this tradeoff to be determined concretely in the

market. When processor time is inexpensive relative to memory, algorithms that use only a little memory but compute for a long time will be favored over those that require more memory but less computation time. The reverse will be true when processor time is expensive relative to memory.

On a single-processor machine in the absence of suspended or recursive procedures, only the process which has control of the processor at that moment will need memory. If the available memory is more than adequate for that process, memory may not be a scarce resource and will therefore carry a price of zero. Furthermore, processes which do not acquire any computation time should not bid for memory (in equilibrium), since they will presumably have no use for it. These intuitions are also reflected in the bidding mechanism described in this section. In the equilibrium on a single processor machine it must be the case that the price of processor time exceeds the maximum bid rate for all but one process. (Otherwise the demand for processors would be greater than the supply of one.[1]) Therefore, for all other processes bidding for memory, the numerator of the maximum bid rate formula will be negative, since those processes would expect to lose money even with zero-cost memory since processor time is already too expensive for them. Since all other competitors enter negative bids (reflecting that they would have to be *paid* for memory usage in order to make a profit), the process that wins the bidding for the processor can enter a bid of zero for the memory that it will use.

However, when memory does become scarce (perhaps because of suspensions or multiple processors sharing a memory space), processes must rent memory space. This establishes proper incentives for memory conservation and allows garbage collection to be based directly on evicting memory "tenants" who are no longer able to afford the going rate. This may be because of outbidding by processes with greater "importance," or because there are no longer any pointers to the information stored in that area.

[1]When there are exact ties at the highest bid level, no equilibrium will exist. This is very similar to the case shown in figure 7-2, which could just as well represent a single-processor scenario with vertical axis rescaled. When this case arises in the RECON system, a winner is chosen arbitrarily. To extend to the memory market, processes which are rejected in this way must explicitly realize to revoke bids on resources which will not be useful without computation time.

Most importantly, rental of memory space results in an economically efficient allocation, offering prioritizing and ensuring the best possible use of space when it becomes a limiting factor.

The memory market presented in this section can also be extended directly to a class of similar problems by redefining the meaning of a unit of memory. In the RECON environment, for example, conflicts can potentially arise between processes that wish to perform revisions on overlapping set of nodes. By considering an individual node as the unit of memory, bidding and awarding of "control" over each node can be performed exactly as for a more finely divided memory market.

7.3 Interruption and Suspension

In a system in which processes may enter bids at any time or where outside "interrupts" must be serviced within the market environment, it is possible that a process using the processor may be outbid before it has completed its desired computation. As mentioned in section 5.5.2, interruption may have significant costs. When atomization (section 7.4.3) is enforced, termination results in a loss of a great deal of effort since no valuable results have yet been produced. Ideally, it would be desirable to avoid these losses by allowing a process to suspend and then resume its computations when the price of processor time has fallen to a level where it is again the high bidder. Of course, suspension also has its costs, but in this case it is possible to assign these costs to the suspended procedure, thus allowing for market trade-offs so as to allocate resources efficiently when suspension and termination are possible.

The mechanism is based directly on the storage rental mechanism described in the previous section. Suspension becomes the default action when an active process is outbid. The suspended process must then pay the rent for the memory locations that it must preserve in order to be able to resume its computation in the future. When memory is not a scarce resource, suspension will have no cost and this will be reflected by a zero

price to preserve necessary memory locations. However, when memory becomes scarce (perhaps because of many suspended processes), the cost of maintaining a suspended state will reflect the true cost of suspension to the system.

When a process is no longer able or willing to pay the rent on its memory locations, it will be terminated. Thus processes are only terminated when it is necessary, in the sense that the costs of suspension now outweigh the benefits. Clearly a process with no funding can no longer pay for its memory locations when suspended. However, there may also come a point when the process projects that it will be less expensive to start the computation over again than to continue to pay for the memory locations needed to remain suspended. The details of how this decision should be made as well as how much to bid for memory while suspended is an important area for further research.

Of course, this model ignores the costs that may be incurred from the actual acts of suspension and restart, but it seems that these costs will normally be negligible. If the costs are in time or in small amounts of additional memory, then the suspended (and later restarted) process can be made to pay for these costs as well.

7.4 Programming Constraints

In addition to the significant considerations for consumers discussed in section 7.1, utilizing the market mechanisms for allocation of computational resources also has an effect on how the individual procedures must be programmed. There are three specific constraints discussed in this section which may have a significant impact on what programming idioms may be used. Defining inputs and outputs and making execution time estimates are required in order to use a market allocation system. The third constraint, which I have called atomization, is a suggestion that can help to ensure that all computations are carried out as efficiently as possible within the market domain.

7.4.1 Knowledge of Inputs and Outputs

In order to participate in the market, a process must know what its required inputs and computed outputs and services are so that it can determine the profit to be gained from the use of computational resources. This is analogous to data-dependency programming used in parallel programming algorithms. In many circumstances it is quite clear what conditions must be satisfied before a particular procedure can execute successfully. However, it may not always be simple to phrase these conditions in the form of required input goods or services.

Since a routine is normally designed with a specific computation or service in mind, the outputs of a particular process should usually be obvious. However, in some cases a process may not be able to guarantee its outputs and will have to consider this risk when making its bids.

Furthermore. there may be a significant problem in describing the goods in the market. Goods and services that are very similar or even substitutable should be treated accordingly. The solution to this problem may lie in defining goods carefully in an object-oriented fashion so that their inter-relationships are manifest. Doyle[12] suggests that a language for the specification of market agents, including the goods themselves, would be a very valuable step in making market-based programming more practical. That suggestion is certainly born out by my research, though I have not investigated design issues for such a language.

For sufficiently complicated systems it may be very cumbersome or perhaps even impossible to accurately define the inputs and outputs of every process. However, for a wide range of tasks, these definitions should be obvious directly from the design of the individual procedures.

7.4.2 Time Estimates

Time estimates for use of computational resources are also necessary to ensure an efficient allocation. Without time information it will be impossible to distinguish between

procedures that perform identical services in vastly different amounts of time. Since rational use of time is a principal goal of market-based allocation, there is no possibility of avoiding this constraint.

Unfortunately, it may be very difficult to offer accurate time estimates for processes. Since in general it is impossible to even determine if a process terminates in a finite amount of time, there may be many domains where precise time estimates are impossible. Of course, time estimate calculations are allowed to be complicated and can take into account many factors of the current state of the computer, but in general it will be impossible to know the exact execution time without actually executing the procedure. If the time required to compute a time estimate is not negligible compared to the actual time of execution, then the processor may be spending its time inefficiently.

For simple procedures, reasonable time estimates can be achieved from a combination of analysis and experimentation. For the RECON system, time estimate formulas are very simple and tend to stay within about 20% of the actual execution time. Whether the estimates are "accurate" is largely a matter of opinion.

However, it appears that very accurate time estimates are not especially critical, at least in the simple domain of the RECON system. Small time estimate mistakes lead to equally small (or sometimes no) misallocations. Inaccuracy of time estimates is a form of uncertainty or incomplete information. Experience in real world markets, which exhibit great uncertainty, shows that good approximations to ideal allocations can still be achieved by markets. In fact, when uncertainty is taken as given, markets (with some restrictions) achieve the best possible allocation from the information available.

In addition, it may be possible for certain learning algorithms to be employed to improve the quality of time estimates based on experience in running the processes. This application of learning algorithms would be an area for significant further research.

There also may be some concern that time estimates be systematically inaccurate, since a process claiming vastly optimistic execution times will get control of the processor more often. However, this behavior will very quickly be discouraged by the market,

since processes which consistently under-estimate execution time will take large monetary losses. Processes may also over-estimate execution time, resulting in increased profits. However, this is a perfectly legitimate strategy that represents a conservative approach to risk taking. Analysis of how to assess and react to risk and then adjust time estimates accordingly is another area requiring significant further research.

7.4.3 Atomization

For use within a market allocation system, there are some advantages to requiring that computations be broken into the smallest possible blocks, or "atomized." In particular, atomization means that any intermediate values that are calculated in series should be considered as separate processes. As a result, individual procedures produces all of their relevant results at the end of execution. The constraint is more difficult to follow with services, since it may not be clear how to break a service into atomic blocks. Atomization is not required to use a market system, but it prevents certain cases in which computations are performed inefficiently. To underscore this point, consider the following example processes:

Process #	Value Computed	Seconds Used
1	a	1
2	b	2
3	c	1
4	d	10

Suppose that in order to compute d, however, the values of a, b, and c must be known. So process 4 also computes the values of a, b, and c, in 2, 3, and 2 seconds respectively, then computes d in 3 seconds.

If the system desires to calculate d (i.e. the good "value of d" has a relatively high price), then its only choice will be to execute process 4, requiring 10 seconds. Ideally, however, we would like to use the methods of processes 1 though 3 to calculate a, b, and

c in a total of 4 seconds, then use the method of process 4 to calculate d for a grand total of 7 seconds.

Requiring atomization will correct the misallocations of the above example. When process 4 is atomized it will be broken into 4 new procedures (named 4a, 4b, 4c, and 4d for easy reference). The goods for the values of a, b, and c are now inputs to 4d. As a result, calculating those values becomes valuable in the market.

When process 1 and process 4a enter bids to calculate a, process 1 will always bid higher since the inputs (none) and outputs (a) are identical, but process 1 requires less time. Similarly, process 2 will beat out 4b and process 3 will beat out 4c. Thus, the market can ensure that d is calculated by the most efficient possible means.

There is one strong caution regarding atomization, however. If the time to run the market and determine the next allocation is significant relative to the running time of the processes, then it may make sense to perform several related tasks as one "process." This avoid using time inefficiently to decide what to do next. Suppose that for the system described in this section the market is evaluated on the same processor and requires 2 seconds. In this case the sequence *market, process 1, market, process 2, market, process 3, market, process 4d* will require 15 seconds, while the sequence *market, process 4* would require only 12 seconds to achieve the same result. This will all be handled correctly, however, if the time estimates are made to include the market evaluation time as suggested in chapter 6. In this case both the atomized versions of process 4 and the original process 4 should be present. Since the time estimates now include market processing time, the market will automatically select the fastest route to calculating d.

7.5 Communication and Scalability

In the modern United States economy, information is a major good which is often more expensive than manufactured goods or services. In large-scale markets containing enormous numbers of agents it is very often difficult, impractical, or impossible to provide

all agents access to all information. As a result agents are often arranged hierarchically, and often agents exist whose sole function is to acquire and distribute information. Presumably, a computational market will also encounter these issues when the number of agents becomes very large.

One of the principal issues involved in handling this problem is determining the means of communication between agents. The Contract Net ([2], also summarized in section 3.1) provides one possible method of communication, specification, and acquiring information. The Contract Net allows for very flexible communication, and offers a means of subcontracting, which may be critical when using decentralized agents in a loosely cooperative manner. However, these mechanisms are somewhat complicated and place additional requirements onto the programmer of the agents involved.

This should not, however, be considered a deficiency in the Contract Net protocol. Rather, it shows that increased exchange of information comes at a price. The RECON system as it now stands offers no direct communication between processes. This is much easier to implement and the design of individual procedures is not much more difficult than in any other programming environment. However, the RECON system is on a very small scale where it is reasonable to have all of the agents have direct access to whatever information they may need to enter bids and perform tasks.

In light of these tradeoffs, the level of communication between processes and the availability of information in general are significant issues in the design of any practical system which is based on market allocations. Extensive continuing research will be necessary to establish guidelines for agent interactions other than simple entering of bids and use of resources.

In addition, when there are very large numbers of agents participating in the economy, it may not be practical for every agent to be directly involved in the market. In real world stock markets, for example, specialized "traders" are usually the only direct participants in the auctions (using WALRAS terminology) for stocks. The traders themselves are not bidding for stock ownership, but are each representing many individuals who wish

to execute trades. Indeed, the very existence of stock markets represents an extension to the basic market mechanisms I have explored in my research.

Thus we see another form of communication which may be required in large systems, as well as the need for hierarchical structure mentioned at the beginning of this section. In the particular example of traders, it appears that much of their information gathering (from potential buyers and sellers) should be performed in parallel, since the traders can all communicate with clients simultaneously with only minimal need for communication between traders. (Of course, traders may still need inter-communication when participating in the auctions.) As the number of agents becomes larger there may be more opportunities for optimization through parallelization.

Others types of information may be valuable as well. An agent which determined the qualities of other agents was mentioned in section 4.3.1, and this concept can be extended to arbitrary information-getters. This information will then be traded just like any other good in the computational market. It will generally be up to the programmer, however, to assess the need for information gatherers and to implement them individually, as well as to design consumers that value information and agents that can put information to use. None of these tasks seem trivial.

Unfortunately, then, scaling a market-based system to include more and more agents may not be as simple as programming more consumers, more producers, and more good specifications. Significant scaling may require significant extensions to the computation market described in this thesis and implemented in my research. The practicality of markets on a large scale is suggested by the success of enormous national economies, though it remains to be proven for computational markets. Indeed, as the nations have moved to being participants in a "world economy" in recent years, the problems of scaling even well-established economic systems have occasionally been manifested.

7.6 Additional Issues in Economics

Economists have studied market systems in great detail. In this section I present a few important considerations from applied economics that I have largely ignored in the relatively simply RECON domain. However, these issues will become more and more likely to play a significant role as the market-based approach is brought to other domains and scaled to larger systems.

7.6.1 Regulation

The regulation of real word markets is normally based on a system of rules, a group of agents which detect and/or judge rule violations, and prescribed penalties for violating these rules. This may be one area where computational markets have a significant advantage over their real-world counterparts. As mentioned in earlier chapters, regulations in a computational economy can be made to be more like physical laws which are impossible to violate. This prevents the need for enforcer agents and penalties.

However, it is still non-trivial to determine what the regulations should be in a computational market. The current RECON implementation, for example, does not allow interruption of any kind because of the possible losses associated (see sections 7.3 and 5.5.2). However, many systems would want to allow interruptions to deal with events beyond the control of the computer itself. Parallel architectures might allow interruptions regularly since results on one processor might make other computations much more valuable than an ongoing computation on another processor.

In a highly evolved computational market, it might be desirable to regulate against monopolies, provide detailed mechanisms for bankruptcy, or disallow the trading of certain kinds of options. As the market becomes increasingly complicated, there may be a need for many of the regulations imposed in the real world. Hopefully experience and experimentation in real-world markets can yield suggestions when a computational market appears to need regulation. Of course, it may also be difficult to tell when regulations

are needed.

In a decentralized system in which many humans are designing agents, and some or all of them (the humans) are interested in only their own goals, it may be possible that agents come into existence that violate regulations. In this case there *can* be a need for enforcers, judges, and prescribed punishments. Even here, however, "fairness" will only be important when violations result in inefficient allocations.

As an example of a potential need for regulation, consider "dishonest" agents which use resources but do not have the funding (or the willingness) to pay for them. Since this kind of "theft" will typically result in inefficient allocations, it is desirable to prevent it. Two obvious solutions are to enforce a significant penalty (such as deletion - remember that there is no concern about cruelty here) against any agent which does not pay debts, or to require advanced payments for any rented resources and deduct the rent from the retainer. There may also be other possible ways of regulating against resource theft. This simple example shows that regulation of large-scale computational markets may be a very complicated issue, and one that (much like real-world markets) will require ongoing research in the future.

7.6.2 Macroeconomic considerations

In addition to the properties of the individual markets, trades, and equilibria that ensure efficient allocations, a long-running economic system will exhibit behaviors that are best examined at a global level. Macroeconomics is a well-established branch of economics which deals with exactly these global issues, including inflation, interest rates, savings and investments, central banks, the supply and meaning of currency, and many others.

It appears to be safe to ignore macroeconomic considerations (as I have done) in small systems with simple agents, but it also seems very likely that these considerations should not be ignored in the development of more evolved computational markets. Especially when the agents become more sophisticated, it is possible that reaction to macroeconomic forces will be possible. For example, if currency is devaluing over time (inflation), then

this should be considered in making rational bids, since the longer the process waits the less it will be able to buy with its money. On the other hand, when there is deflation, there may be incentives to hold capital and then spend it when it has more purchasing power in the future.

In the simple RECON implementation, the money supply is not intentionally regulated, but does change significantly because new consumers enter the market and bring along their endowments. As a result, even this domain exhibits macroeconomic fluctuations, especially inflation since the money supply will normally increase over time. However, agents are not programmed to react to inflationary trends but rather to optimize their utility/profit at each stage without consideration to the past or future.

Macroeconomics offers a wealth of theory that can be brought to bear against these issues, and also can suggest rational strategies for agents to respond to these global changes. To date, however, macroeconomics has been almost completely ignored in the study of applying economics to computer science.

7.6.3 Externalities

Because processes typically affect the "world" in more ways than simply producing outputs, there may be some side effects whose value is very difficult to account for. These side effects are called externalities in economics, and are common in real-world markets. Externalities are usually partitioned into *negative externalities*, meaning that a side effect has a net negative effect on other agents in the economy, and *positive externalities*, with the symmetric meaning.

Example from ordinary markets of negative externalities include pollution from production, large ugly signs out front of a store, and poorly-maintained houses in a neighborhood. All of these affect not only themselves but other agents in the economy who typically have no control over the offending agent. Positive externalities might arise from teaching, invention, or exceptionally well-maintained houses in a neighborhood. In each case outside agents are affected positively in a way that they have not completely paid

for.

Externalities generally cause significant difficulty in market systems, since it may be very hard to assign value (or penalty) to them appropriately. To provide ideal incentives and resource allocation, the person who maintains a house extremely well should actually be paid by his neighbors, since they are direct beneficiaries. The issue of handling externalities in general is a significant area of ongoing research in economics, and will also have to be researched within the computational market domain. In this section I do not provide the answers, but suggest places where externalities may arise in computational markets, along with some general ideas for handling them rationally.

7.6.3.1 Positive Externalities

The most obvious class of positive externalities in a computational market are those procedures whose executions allows other procedures to run faster in the future. Many common processes such as garbage collection and defragmentation of disk space can have such an effect. Each of these processes allows faster accesses in the future, thereby accelerating future computations. Another example with a similar effect is a process which tabulates a function. Then any process that needs values of that function in the future can simply look them up rather than recalculating them.

Each of these processes may become valuable in itself if consumers value its execution. However, it is not obvious how to account for the total benefits of procedures with positive externalities, especially since the benefits of their execution may be reaped repeatedly and indefinitely into the future. One reasonable possibility is to require a sort of charge-per-use system which is a simplified version of a patent or royalties. Thus, for example, each time a process used a tabulated function it would pay the process which originally tabulated it. This may help to provide proper incentives for tabulation, though it is still difficult to guess the value accurately, since future uses cannot be fully anticipated. Unfortunately, this scheme has a significant drawback. When an agent is forced to make the choice between paying for the fast, tabulated function and paying for the

extra computation time to recalculate the value, it may sometimes be more profitable to recalculate. However, since the function has *already been* tabulated, it is inefficient to ever recalculate values.

7.6.3.2 Negative Externalities

Although computational markets can be built in such a way as to make many negative externalities be impossible, there are still cases where the execution of a process will somehow have a negative impact on other processes. One obvious case is that of disk fragmentation, mentioned in the discussion of positive externalities. When a process performs alternate writing and deleting operations on a disk, the file system may become heavily fragmented with various pointers telling where to look for the next part of the file. In general, this fragmentation will slow down disk accesses, and therefore have a negative affect on any process which performs a read operation in the future.

One possible method of handling negative externalities within the market framework is to require that processes somehow pay for any negative effects that they may have. This establishes an appropriate disincentive for negative externalities. These "fines" can then be directed to paying processes that perform services to correct the problems, when that is possible. In the example at hand, processes would be charged for each delete or write operation that caused fragmentation and that money would be used to bid for the service of defragmentation. When "blame" is easy to assign and processes exist that have positive externalities canceling out the negative externalities, this sort of earmarked fining protocol seems appropriate. However, a formula still needs to be developed to determine the magnitude of fines.

7.6.4 Complementarity

In many markets there may be goods whose functionality is somehow tied together. These complementary goods, as they are called in economics, typically rely on each other in order to be useful. Real world markets are filled with such goods: tennis rackets and

tennis balls; cameras and film; fishing rods, reels, line, and hooks. A fishing rod is not very useful without its complements, at least if the intent is to catch fish. If fishing line suddenly became prohibitively expensive, then the demand for fishing rods would presumably fall, since they are not worth much by themselves.

This is often given as the defining characteristic of complementarity, specifically that price increases in one good actually result in a *drop* in demand for the good's complements. This is slightly counterintuitive for the kinds of markets discussed in this thesis. Normally, we are interested in trade-offs, which mean that when a good is very expensive, other goods can be substituted. Goods for which price increases in one cause demand increases in the other are called substitutable, which seems to be the common case in simple computational markets.

Unfortunately, complementarities can cause significant problems to the system. In particular, the iterative convergence methods used by WALRAS to calculate equilibrium prices are not guaranteed to converge in the case of complementarities. It remains to be seen if computationally tractable methods can be developed that solve markets that are not restricted in this way.

In light of this, it is important to determine if complementarities are likely to exist in a computational market. Unfortunately, there are several simple cases where complementarities seem likely to arise. One specific case is the relationship between memory and processor time. To a certain extent these goods are substitutable, since tradeoffs can be made between algorithms that are space-intensive but time-efficient and those that are time-intensive but space-efficient. However, there is also a complementarity that exists at extreme prices. If memory becomes extremely expensive, then demand for processor time will *decrease*, since most processes will not be able to do anything useful with processor time but no memory.

There is also a more general class of complementarities that can arise from complements in inputs. Suppose for example that a process requires two values, x and y, and produces a value z. Further suppose that x is a common input requirement, but that

y is only needed to calculate z, though this is not absolutely necessary to exhibit complementarity. If many other processes bid up the value of calculating x because it is a commonly needed input, then the demand for y may actually fall, since y is not valuable if x is too expensive to make calculating z be worthwhile.

Whenever inputs take the form of a conjunctive expression, it is at least conceivable that complementarities can result. It remains to be seen how significant of a problem this actually is in practice and if small patches can be used to correct for occasional complementarities.

7.7 Extension to Parallel Processing Environments

Although parallel processing environments are bound to carry many implementation complications, the extension of processor markets to this domain is straightforward at a theoretical level. The auction mechanism remains nearly identical to that of section 5.5.2 and section 6.2, with the only extension being that the supply is altered from one processor to however many identical processors are available. If the processors are differentiable, then each type should be treated as a separate goods with its own auction.

The bidding side requires only two simple extensions, namely that bids be given on a per unit time per processor basis and that demand "curves" indicate the number of processors that a particular agent needs to use to perform its computation. These extensions are nearly identical to those proposed in section 7.2 for a market in memory space. Specifically the maximum bid rate formula for processor time in the parallel computation environment becomes:

$$maximum\ rate = \frac{value\ of\ outputs\ and\ services - value\ of\ all\ other\ inputs}{(estimated\ execution\ time)(estimated\ processors\ required)}$$

As usual, the demand curve will be zero for all prices above the maximum bid rate, and equal to the number of processors needed for any price below the maximum bid rate. There is also the possibility that a process can use a variable number of processors to

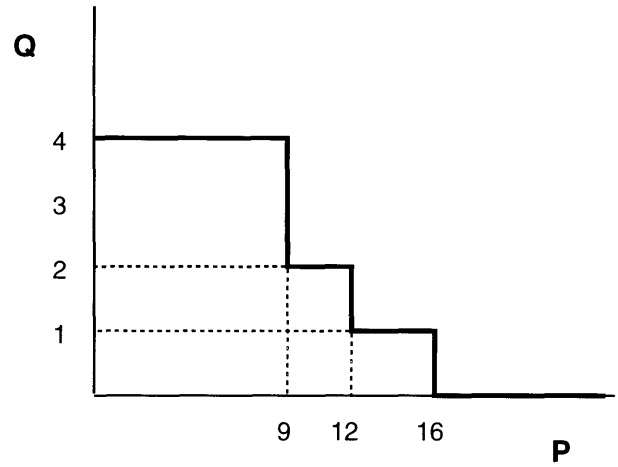


Figure 7-1: Demand Curve for a Process with Three Ways of Using Processors.

perform its computation. In this case it will presumably be faster to use more processors than fewer. The resulting demand curve will be a staircase that changes height at each of the zero-profit points. Specifically, there will be a maximum bid rate for each possible number of processors that the process could use. To clarify this concept, suppose that a particular process has an expected profit (the numerator in the maximum bid rate formula) of \$144. It has three different ways of using processors that result in the following expected execution times: 4 processors for 4 seconds, 2 processors for 6 seconds, or 1 processor for 9 seconds. Using the formula above, we see that it will be willing to use 4 processors when the price (per unit time per processor) is less than \$9 (from $\frac{144}{(4*4)}$), 2 processors when the price is less than \$12, and and 1 processor when the price is less than \$16. The resulting demand curve is shown in figure 7-1.

Given these extensions, the market discovers the equilibrium prices by the usual method: it aggregates the demand from all bidding processes and finds the intersection of the aggregate demand with the aggregate supply (the supply is simply a horizontal line at the level of the number of processors available). The resulting equilibrium indicates the price of processor time, and all processes which are willing to compute at that price can be awarded the number of processors they have demanded.

One difficulty in parallel architectures is that there may be a great deal of asynchronic-

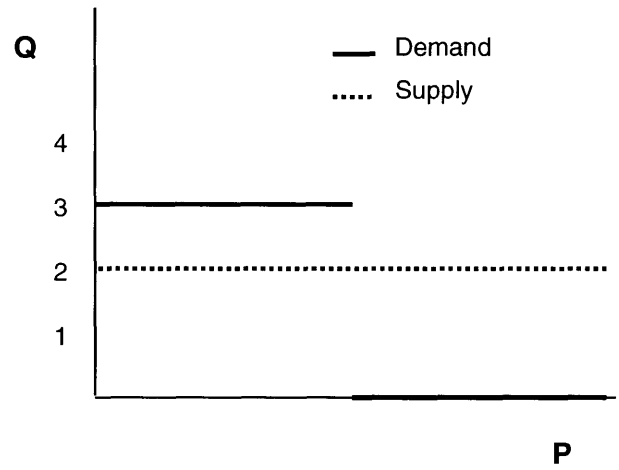


Figure 7-2: Processor Market with No Equilibrium.

ity in the use of processors. It will certainly not normally be the case that all processors are rented and freed at exactly the same times. From the point of view of market design, this is no problem — the aggregate supply curve will simply represent the number of processes available in the particular auction. The problem is that agents may need a certain minimum number of processors in order to operate. When that number is greater than the number available, the equilibria may not make very much sense. Imagine that there is only one process bidding and it requires 3 processors, but only 2 are available. The aggregate demand-aggregate supply diagram of figure 7-2 results.

Because of the discontinuity in demand, there is no equilibrium value — the two curves do not intersect. This makes sense intuitively, since the process cannot get what it wants (3 processors) and the supplier cannot sell what it has (2 processors). Most likely the logical action in the case of no equilibrium is to do nothing. Then when more processors become available an equilibrium will result and the market will properly allocate resources. In a practical implementation, failure to find an equilibrium should probably be interpreted as though none existed, though this might not necessarily be the case.

In addition, this methodology generally favors processes which can use a smaller amount of processors, since they can keep buying up small quantities out from under

the processes which are waiting around for a larger number to be free. Thus processes requiring large numbers of processors may have to pay to rent processors that are not actually being used so as to eventually acquire enough processors to perform the desired computation. Therefore the remedy to this problem will be to bid on however many processors are available using a time estimate which includes the waiting time for enough other processors to become free. Since every running process has entered a time estimate itself, reasonably accurate information may be available to use in making these estimates. This method will ensure efficient allocation since it always ensures that the processors are used for maximum expected future benefit, even when that may require a processor sitting idle.

Finally, there is the possibility of differentiated processors. However, this should not cause any confusion, at least at a theoretical level. A process may bid on any processor that is capable of performing the required tasks. The maximum bid rate will be calculated separately for each processor using specialized time estimates. Thus a process will always be willing to bid more for a processor that can perform its task faster. As usual, the resultant demand for each good will be a function of other goods' prices as well, namely the prices of the other processors. The final demand function represents a demand for only the most desirable processor whose price is below the maximum bid rate for that processor. The market equilibrium will automatically serve as a match-maker. Each processor will be allocated in an efficient manner, maximizing the benefit from its use.

Chapter 8

Conclusions

As the complexity of computational environments increases, the utility of decentralized allocation mechanisms will become evident. This thesis combined research in economics, decision theory, and computer science to provide a market-based means of allocating the processor, and suggested ways to allocate similar resources. These bidding and auction mechanisms are a fundamental step in developing market-based environments.

The processor auction mechanism detailed in section 5.5.2 satisfies virtually all of the goals for a computation market outlined in section 4.3, with the possible exception of rapid execution, though this failure should be easily overcome by software optimization and/or specialized hardware. The processor rental market allows for efficient allocation of *time* with respect to other resources, goods, and services, and provides an implicit "rational" decision, in the sense of limited rationality of the meta-greedy assumption [24]. The method of taking bids and using them to determine prices is very simple. The combination of the reasoning economy with the processor market allows for rapid dynamic adjustment without necessarily requiring explicit replanning.

The mechanisms provided are relatively straightforward to implement and can be programmed in a computationally tractable manner. Although this research is only the very beginning of the development of practical market-based resource allocation systems for computation, I am hopeful that I have laid some of the groundwork for further

research. The advantages of markets and the extent of economic research and experience point to the eventual goal that large-scale market-based allocation systems some day be a part of practical computation environments.

Bibliography

[1] John S. Breese and Eric J. Horvitz. Ideal reformulation of belief networks. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 64–72, July 1990.

[2] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983.

[3] Johan de Kleer, Jon Doyle, Guy L. Steele Jr., and Gerald Jay Sussman. AMORD: Explicit control of reasoning. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, pages 116–125, 1977.

[4] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, 1988.

[5] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(2):231–272, 1979.

[6] Jon Doyle. The ins and outs of reason maintenance. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 349–351, 1983.

[7] Jon Doyle. Constructive belief and rational representation. *Computational Intelligence*, 5(1):1–11, February 1989.

[8] Jon Doyle. Rational control of reasoning in artificial intelligence. In André Fuhrmann and Michael Morreau, editors, *The Logic of Theory Change*, volume 465 of *Lecture Notes in Artificial Intelligence*, pages 19–48. Springer-Verlag, Berlin, 1990.

[9] Jon Doyle. Rational belief revision (preliminary report). In Richard E. Fikes and Erik Sandewall, editors, *Proceedings of the Second Conference on Principles of Knowledge Representation and Reasoning*, pages 163–174, San Mateo, CA, 1991. Morgan Kaufmann.

[10] Jon Doyle. Rationality and its roles in reasoning. *Computational Intelligence*, 8(2):376–409, 1992.

[11] Jon Doyle. Reason maintenance and belief revision: Foundations vs. coherence theories. In Peter Gärdenfors, editor, *Belief Revision*, pages 29–51. Cambridge University Press, Cambridge, 1992.

[12] Jon Doyle. A reasoning economy for planning and replanning. In *Technical Papers of the ARPA Planning Initiative Workshop*, 1994.

[13] D. Ferguson, Y. Yemini, and C. Nikolaou. Microeconomic algorithms for load balancing in distributed computer systems. In *Proceedings of the IEEE International Conference on Distributed Computer Systems*, pages 491–499, 1988.

[14] Peter Gärdenfors. The dynamics of belief systems: Foundations vs. coherence theories. *Revue Internationale de Philosophie*, 172:24–46, January 1990.

[15] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman, 1979.

[16] Eric J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *Proceedings of the Third AAAI Workshop on Uncertainty in Artificial Intelligence.* AAAI, 1987.

[17] J.O. Kephart, T. Hogg, and B.A. Huberman. Dynamics of computational ecosystems. *Physical Review A*, 40:404–421, 1989.

[18] Barton L. Lipman. How to decide how to decide how to ...: Modeling limited rationality. *Econometrica*, 59(4):1105–1125, July 1991.

[19] T. W. Malone, R. E. Fikes, K. R. Grant, and M. T. Howard. Enterprise: A market-like task scheduler for distributed computing environments. In B. A. Huberman, editor, *The Ecology of Computation*, pages 177–205. North-Holland, Amsterdam, 1988.

[20] M. S. Miller and K. E. Drexler. Comparative ecology: A computational perspective. In B. A. Huberman, editor, *The Ecology of Computation*, pages 51–76. North-Holland, Amsterdam, 1988.

[21] M. S. Miller and K. E. Drexler. Incentive engineering for computational resource management. In B. A. Huberman, editor, *The Ecology of Computation*, pages 231–266. North-Holland, Amsterdam, 1988.

[22] M. S. Miller and K. E. Drexler. Markets and computation: Agoric open systems. In B. A. Huberman, editor, *The Ecology of Computation*, pages 133–176. North-Holland, Amsterdam, 1988.

[23] P. Mongin and B. Walliser. Infinite regressions in the optimizing theory of decision. In B. Munier, editor, *Risk, Decision and Rationality*, pages 435–457. D. Reidel Publishing, Boston, 1988.

[24] Stuart Russell and Eric Wefald. On optimal game-tree search using rational meta-reasoning. In *Artificial Intelligence: Proceedings of the 11th International Joint Conference (IJCAI-89)*, pages 334–340, 1989.

[25] Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial Intelligence*, 49:361–395, 1991.

[26] Stuart J. Russell, Devika Subramanian, and Ronald Parr. Provably bounded optimal agents. In *Artificial intelligence: Proceedings of the 13th International Joint Conference (IJCAI-93)*, pages 338–344, 1993.

[27] R.G. Smith. The contract net protocol: high-level communication and control in a distributed problem solver. *IEEE Trans. on Computers*, C-29, December 1980.

[28] I. E. Sutherland. A futures market in computer time. *Communications of the ACM*, 11(6):449–451, June 1968.

[29] Carl A. Waldspurger. A distributed computational economy for utilizing idle resources. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, Cambridge, Massachusetts, 02139, May 1989.

[30] Michael P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[31] S. Winter. Optimization and evolution in the theory of the firm. In R. H. Day and T. Groves, editors, *Adaptive Economic Models*, pages 73–118. Academic Press, New York, 1975.

[32] Shlomo Zilberstein. *Operational Rationality Through Compilation of Anytime Algorithms*. PhD thesis, University of California at Berkeley, May 1993.